Zweistufige, Planer-basierte Organic Computing Middleware

Dissertation

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

der Fakultät für Angewandte Informatik

der Universität Augsburg

eingereicht von

Julia Schmitt

Julia.Schmitt@informatik.uni-augsburg.de

15. April 2013

Erstgutachter: Prof. Dr. Theo Ungerer

Zweitgutachter: Prof. Dr. Bernhard Bauer

Tag der mündlichen Prüfung: 11.03.2013

Zusammenfassung

Mit fortschreitender Entwicklung neuer Software- und Hardware-Komponenten steigt die Komplexität von Computersystemen. Die Installation und Wartung moderner Systeme ist schwierig und aufwändig. Die *Autonomic Computing* und *Organic Computing* Initiativen haben Methoden entwickelt, um diese Komplexität zu beherrschen. Sie bedienen sich dazu Konzepten aus der Natur und identifizierten vier Eigenschaften, die zukünftige Systeme haben sollen. Diese Eigenschaften sind Selbst-Konfiguration, Selbst-Optimierung, Selbst-Heilung und Selbst-Schutz und wurden in der Middleware OCμ umgesetzt.

Jede der Selbst-X Eigenschaften wurde ursprünglich isoliert entwickelt. Allerdings bestehen zwischen den Eigenschaften Abhängigkeiten. Wenn beispielsweise die Selbst-Heilung nach einem Fehler das System repariert, die Selbst-Optimierung jedoch eine bessere Konfiguration findet, erhöht sich die Anzahl notwendiger Eingriffe in das System. Zudem sammelte jede Selbst-X Eigenschaft die für sie notwendigen Daten selbst.

Um Synergien zwischen den verschiedenen Selbst-X Eigenschaften nutzen zu können und Wechselwirkungen auszuschließen, wurden in dieser Arbeit neue Konzepte der Selbst-Organisation entwickelt und in OCμ implementiert und evaluiert. Dazu wurde die OCμ Architektur weiterentwickelt sowie ein neuer Organic Manager als zentrales Steuerelement entworfen und implementiert. Um alle Selbst-X Eigenschaften gemeinsam umsetzen zu können, wurde ein *automatischer Planer* in OCμ integriert. Für ihn wurden in dieser Arbeit mehrere Planungsmodelle entwickelt und implementiert. Als Ergänzung dient ein Reflex Manager, der Pläne speichert und dadurch die Reaktionszeit des Systems verringern kann. Er ist fähig Pläne wieder zu verwenden, die ursprünglich für einen anderen, ähnlichen Systemzustand erstellt wurden. Für die Ausführung der Pläne wurde ein Actuator entwickelt und implementiert. Er kann konkurrierende Pläne, die aus der Konkurrenz von Reflex Manager und Planer entstehen, auflösen.

Mit Hilfe von Cloud-Computing-Szenarien wurden verschiedene Planungsmodelle evaluiert. Der Planer kann auch bei häufigen Störungen die Selbst-Konfiguration, Selbst-Optimierung und Selbst-Heilung sowie weitere Ziele der Anwender umsetzen. Zudem wird gezeigt, wie verschiedene Parameter sich auf das Verhalten des Reflex Managers auswirken.

Vorwort

Die vorliegende Arbeit entstand in den Jahren 2009 bis 2012 während meiner Tätigkeit als wissenschaftliche Mitarbeiterin am Lehrstuhl für Systemnahe Informatik und Kommunikationssysteme an der Universität Augsburg. Zusammen mit Michael Roth war ich im Rahmen des DFG-Schwerpunktprogramms 1183 Organic Computing angestellt. Die bereits existierende Middleware OCμ wurde angepasst, um Selbst-Konfiguration, Selbst-Optimierung und Selbst-Heilung durch einen Planer umsetzten zu können. Meine Aufgabe umfasste die Integration und Evaluierung eines automatischen Planers, sowie die Konzeption und Implementierung eines Reflex Managers und Actuators.

Mein aufrichtiger Dank gebührt meinem Erstbetreuer Prof. Dr. Theo Ungerer für seine stetige Unterstützung und Inspiration. Außerdem danke ich Herrn Prof. Dr. Bernhard Bauer für die Übernahme des Korreferats. Meinen Kollegen Rolf Kiefhaber und Michael Roth aus dem Organic-Computing-Bereich danke ich für die vielen fruchtbaren Diskussionen und ihre Hilfe bei der Programmierung von OCμ. Für die Korrektur meiner verschiedenen Arbeiten möchte ich meinem Kollegen Florian Kluge meinen Dank aussprechen.

Meiner Familie und meinen Freunden danke ich für die gemeinsamen Stunden, die für den entsprechenden Ausgleich gesorgt haben, der ebenso für das Gelingen meiner Dissertation notwendig war.

Julia Schmitt

Inhaltsverzeichnis

1 Einleitung

Die Komplexität von Computersystemen wächst ständig. Die Ursache liegt in der steigenden Anzahl und der Heterogenität der beteiligten Komponenten. Zudem wachsen auch die Anforderungen und Fähigkeiten der einzelnen Komponenten, sowie die Anforderungen an das Gesamtsystem.

IBM entwickelte als Lösungsansatz das *Autonomic Computing* (AC). Horn [11] orientierte sich am menschlichen Nervensystem und verwendete es als Vorbild um zu zeigen, wie ein komplexes System gesteuert werden kann. Kephart et al. [13] definierten vier Eigenschaften, die ein AC System haben sollte: *Selbst-Konfiguration*, *Selbst-Optimierung*, *Selbst-Heilung* und *Selbst-Schutz*. Autonomic Computing konzentriert sich sehr stark auf Anlagen mit vielen Servern, wie beispielsweise ein Data-Center.

Das DFG Schwerpunktprogramm 1183 Organic Computing (OC) [1] hingegen umfasst auch eingebettete Komponenten. Im Organic Computing organisieren sich die Computersysteme selbst, so dass der Administrationsaufwand sinkt. Am Lehrstuhl Systemnahe Informatik und Kommunikationssysteme in Augsburg wurde im Rahmen des OC Projekts eine Service-orientierte Middleware entwickelt, die Techniken des Organic Computing implementiert. Eine Middleware kann die Heterogenität der verwendeten Komponenten verdecken und führt zu einer homogenen Umgebung für darauf aufsetzende Anwendungen.

Innerhalb dieser Middleware wurde jede der vier Selbst-X Eigenschaften als einzelner Service konzipiert. Jeder dieser Services erhebt die für ihn notwendigen Daten selbst. Da jede dieser Selbst-X Eigenschaften isoliert entwickelt und evaluiert wurde, können Wechselwirkungen entstehen. Beispielsweise wählt die Selbst-Konfiguration einen geeigneten Knoten für einen Service aus und startet ihn dort. Die Selbst-Optimierung jedoch versucht eine ausgeglichen Last herzustellen und verschiebt diesen Service auf einen anderen Knoten.

Eine gemeinsame Umsetzung der Selbst-X Eigenschaften kann solche Wechselwirkungen verhindern und Synergien schaffen. So kann eine gemeinsame Datenbasis die doppelte Datenerhebung verhindern. Synergien können entstehen, wenn beispielsweise die Selbst-Optimierung nicht mehr nötig ist, weil sich durch Maßnahmen der anderen Selbst-X Eigenschaften bereits ein optimales System gebildet hat.

Um Konflikte zu verhindern und die Synergien zu nutzen werden in dieser Arbeit Selbst-Konfiguration, Selbst-Heilung und Selbst-Optimierung gemeinsam implementiert. Als Werkzeug dient ein automatischer Planer. Er verwendet Aktionen mit Vor- und Nachbedingung und berücksichtigt somit die Effekte einer Aktion im voraus.

Als Ergänzung des Planers dient ein Reflex Manager. Wie bei einem Menschen, der gewisse Verhaltensweisen ohne vorheriges Nachdenken ausführen kann, soll auch der Reflex Manager das Verhalten des Planers lernen und anwenden. Er kann lange Planungszeiten überbrücken und verkürzt dadurch die Reaktionszeit des Systems. In dieser Arbeit wird ein Reflex Manager vorgestellt, der Pläne speichert. Er kann Pläne wiederverwenden, wenn das System einen bereits bekannten oder einen hinreichend ähnlichen Zustand erreicht hat.

Der Planer wird mit verschiedenen Modellen evaluiert. Unter anderem wird gezeigt, dass er die Selbst-X Eigenschaften zuverlässig umsetzten kann. Zudem können zusätzliche Ziele von Anwendern integriert werden.

Diese Arbeit ist wie folgt aufgebaut: Kapitel 2 beschreibt die Grundlagen des Autonomic und des Organic Computing. Zudem werden die Grundlagen eines automatischen Planers geklärt. Abschließend gibt es einen Überblick über verwandte Ansätze und Unterschiede zu den hier vorgestellten Konzepten. Kapitel 3 beschreibt die historische und die aktuelle Architektur eines OCμ Knotens. Der neue Organic Manager wird in Kapitel 4 erklärt. Er beinhaltet die Integration des Planers, sowie zwei seiner Modelle und vergleichende Evaluierungen zu ihnen. Zudem werden der Reflex Managers und der Actuator eingeführt. Kapitel 5 beinhaltet eine ausführliche Evaluierung eines kompletten OC Systems mit allen durch den Planer bereitgestellten Selbst-X Eigenschaften. Kapitel 6 umfasst ein erweitertes Modell mit ausführlichen Evaluierungen. Die Auswirkungen des Ausfalls einzelner Knoten und eines geplanten Ausfalls im Rahmen einer Wartung werden in Kapitel 7 untersucht. Zudem werden die Parameter des Reflex Managers evaluiert. Abgerundet wird diese Arbeit durch eine Zusammenfassung und Ausblick in Kapitel 8.

2 Grundlagen

In diesem Kapitel werden die Grundlagen der Arbeit beleuchtet. Während das *Autonomic Computing* seinen Ursprung in Amerika hat, entwickelte sich *Organic Computing* in Deutschland. Beide Bereiche werden nachfolgend erklärt. Des Weiteren wird in dieser Arbeit ein automatischer Planer verwendet, dessen Konzept erklärt wird. Abschließend finden sich verwandte Arbeiten.

2.1 Autonomic Computing

Als eine der großen Herausforderungen zukünftiger Rechenzentren nannte IBM die *Komplexitäts-Krise*. Die ständig steigende Zahl an Software-Anwendungen führt zu einem hohen Verwaltungs- und Wartungsaufwand. Als Lösung schlug Horn [11] vor, dass sich die Systeme selbst verwalten sollen. Um dies zu erreichen, definierte er mehrere Eigenschaften, die ein AC-System besitzen sollte. Kephart und Chess [13] extrahierten daraus vier Selbst-X Eigenschaften, welche sowohl im Autonomic als auch im Organic Computing eine zentrale Stellung einnehmen.

- **Selbst-Konfiguration:** Große Datenzentren werden von verschiedenen Anbietern genutzt und können unterschiedliche Dienste anbieten. Die Installation, Konfiguration und Wartung solcher Systeme ist zeitaufwändig und fehleranfällig. Die Selbst-Konfiguration soll diese Abläufe automatisieren. Dem System werden Ziele vorgegeben, die es selbstständig erfüllt.

- **Selbst-Optimierung:** Moderne Systeme bieten viele Einstellmöglichkeiten, die jedoch manuell vorgenommen werden müssen. In einem AC-System suchen die teilnehmenden Komponenten selbstständig ständig nach Verbesserungsmöglichkeiten für ihre Leistung und Effizienz.

- **Selbst-Heilung:** Die Suche nach Ursachen von Problemen in großen, komplexen Systemen kann sehr zeitraubend sein. AC-Systeme entdecken, analysieren und reparieren Probleme selbstständig.

- **Selbst-Schutz:** Angriffe und kaskadierende Fehler müssen manuell erkannt und behoben werden. Ein AC-System verteidigt sich selbständig gegen An-

griffe. Zudem warnt es den Administrator rechtzeitig, um Ausfällen vorzubeugen.

Um diese Selbst-X Eigenschaften umzusetzen, schlagen Kephart und Chess eine MAPE-Architektur vor. MAPE ist die Abkürzung für die vier Phasen eines Autonomic Managers: *Monitor, Analyze, Plan, Execute*. Abbildung 2.1 zeigt die Struktur eines AC-Elements. Jedes Element verfügt über einen Autonomic Manager. Er beobachtet das Element und analysiert die Daten. Anschließend erzeugt er einen Plan und führt diesen aus. Alle vier MAPE-Komponenten haben dabei Zugriff auf ein gemeinsames Basiswissen.

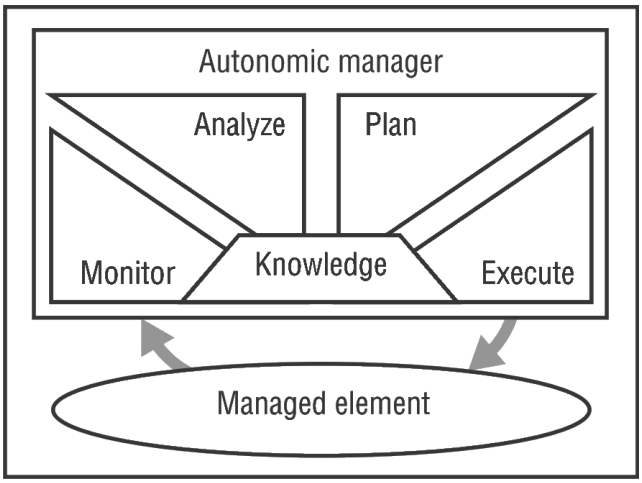


Abbildung 2.1: MAPE Zyklus in Autonomic Computing [13]

Als *Managed Element* bezeichnen Horn und Chess alle Elemente aus normalen Systemen. Beispielsweise können Hardware-Ressourcen wie Speicher, CPU oder Drucker aber auch Software wie Datenbanken oder gewachsene Altsysteme von einem Autonomic Manager kontrolliert werden. Aber auch größere Einheiten wie ein Anwendungs-Software oder eine Firma können durch einen Autonomic Manager verwaltet werden. Die Autonomic Manager kommunizieren untereinander, um systemweite Ziele umzusetzen.

2.2 Organic Computing

Ebenso wie beim Autonomic Computing geht auch das Organic Computing von einer steigenden Komplexität aus. Während Autonomic Computing hauptsächlich Datenzentren betrachtet und homogene Komponenten voraussetzt, betrachtet *Organic Computing* ein größeres Anwendungsgebiet. Unter anderem werden heterogen Komponenten und eingebettete Systeme berücksichtigt.

2003 wurde Organic Computing in einem Positionspapier [4] vorgestellt. Ein OC System wird dabei als ein selbst-organisierendes System beschrieben, dass sich seiner Umgebung anpasst. Es ist selbst-konfigurierend, selbst-optimierend, selbst-heilend und selbst-schützend. Die Ausprägung dieser Eigenschaften sind je nach Anwendungsgebiet unterschiedlich.

Die AC und OC Initiativen unterscheiden sich auch in ihren Zielen. Im AC werden Systeme betrachtet, die sich selbst verwalten, während im OC selbst-organisierende Systeme betrachtet werden. Selbst-Organisation ist jedoch keine der Selbst-X Eigenschaften. Sie ist vielmehr ein übergeordnetes Phänomen, das aus der Umsetzung der Selbst-X Eigenschaften entsteht.

Aus der Selbst-Organisation kann auch Emergenz entstehen. Emergenz ist ein global beobachtbares Verhalten, das sich nicht aus dem Verhalten Einzelner ableiten lässt. Ameisen erzeugen beispielsweise eine Ameisenstraße, wenn sie auf Futtersuche sind. Müller-Schloer [23] versucht ein OC System mit Hilfe kontrollierter Emergenz zu steuern, um unerwünschte emergente Effekte zu vermeiden.

Richter et al. [32] entwickelten eine generische Architektur um den Ansprüchen des Organic Computing gerecht zu werden. Herzstück ist dabei die in Abbildung 2.2 dargestellte Observer/Controler Architektur. Unten befindet sich das zu beobachtende System (SuOC: System under Observation and Control). Ein *observer* beobachtet das System und leitet seine Informationen an einen *controller* weiter. Dieser wählt Aktionen und beeinflusst das System.

Diese Regelschleife erinnert stark an den MAPE-Zyklus. Der *observer* übernimmt die Beobachtung des Systems und analysiert die beobachteten Daten(Monitor, Analyze). Der *controller* entscheidet welche Aktionen ausgewählt werden und führt sie aus (Plan, Execute). Wie in einem AC System kann auch hier der Anwender dem System Ziele vorgeben.

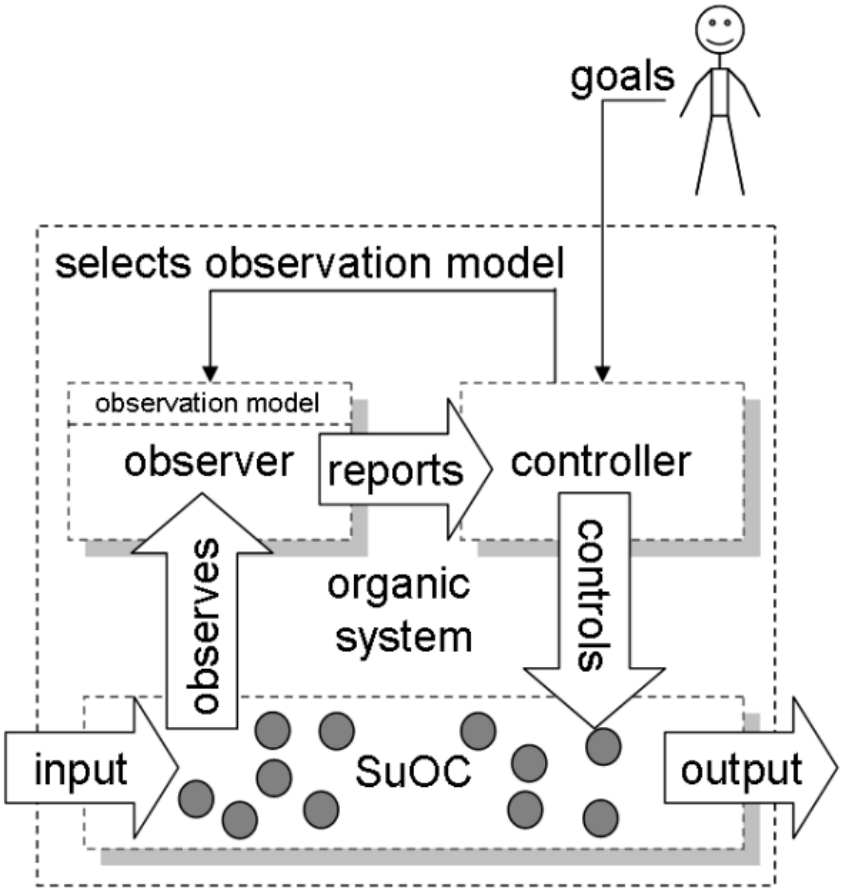


Abbildung 2.2: Observer Controller Architektur in Organic Computing [32]

2.3 Automatisches Planen

Automatisches Planen ist ein Verfahren aus dem Bereich der künstlichen Intelligenz. Ein automatischer Planer erstellt eine Folge von Aktionen, um einen Ausgangszustand in einen Zielzustand zu überführen. Diese Folge von Aktionen wird Plan genannt.

Einen Planungsprozess kann man sich als Graphensuche vorstellen. Die Knoten entsprechen dabei den möglichen Zuständen und die Pfeile Aktionen um von einem Zustand in einen anderen zu gelangen.

Anfangs konnten die Planer nur Zustände verarbeiten, die aus Booleschen Werten bestanden. In der Zwischenzeit haben sich die Planer in verschiedene Bereiche aufgespalten. Ein klassisches Planungsproblem liegt vor, wenn folgende Bedingungen erfüllt sind:

- Endliche Anzahl von Zuständen und Aktionen
- Vollständige Beobachtbarkeit der Zustände
- Deterministische Aktionen
- Statisches System
- Vorgegebene Menge von Zielzuständen
- Sequentielle Pläne
- Implizite Zeit
- Offline Planung

Eine Aktion ist deterministisch, wenn sie aus einem Zustand genau einen Folgezustand erzeugt. Ist diese Bedingung nicht erfüllt, spricht man von probabilistischer Planung. Ein statisches System liegt vor, wenn Zustandsänderungen nur aufgrund von Aktionen geschehen. Bei klassischen Planungsproblemen wird die Zeitdauer einer Aktion nicht berücksichtigt, stattdessen gibt es nur eine Folge von nacheinander auftretenden Zuständen. Offline Planung bedeutet in diesem Zusammenhang, dass Zustandsänderungen, die während des Planungsprozesses auftreten, nicht berücksichtigt werden.

2.3.1 PDDL Syntax

Die *Planing Domain Definition Language* (PDDL) ist eine Standardsprache um automatische Planer anzusprechen. Sie entwickelte sich aus STRIPS [6] und ADL [27]

und wird heutzutage von den meisten Planern unterstützt. Mit Hilfe von PDDL können komplexe Planungsaufgaben beschrieben werden, jedoch unterstützen die meisten Planer nur Teile von PDDL. In PDDL werden Schlüsselwörter verwendet, im Folgenden werden die wichtigsten erläutert.

objects: Objekte auf die sich die Planung bezieht, z. B. Services

types: Arten von Objekte

predicates: Eigenschaften von Objekten, angegeben in Booleschen Werten

functions: Numerische Eigenschaften von Objekten

action: Können die Eigenschaften von Objekten verändern

init: Initialer Zustand des Systems

goal: Zielzustand des Systems

Ein Zustand wird durch die Belegung der *predicates* und *functions* definiert. Aktionen können eine oder mehrere dieser Variablen ändern und dienen dazu, einen Zustand in einen anderen zu überführen.

Der Startzustand enthält die verfügbaren Objekte und alle ihre numerischen Eigenschaften. Boolesche Werte werden meist mit falsch initialisiert, wenn sie nicht explizit auf wahr gesetzt werden. Der Zielzustand enthält die Eigenschaften mit ihren gewünschten Belegungen. Sämtliche im Zielzustand nicht enthaltenen Eigenschaften können einen beliebigen Wert annehmen.

Um ein Planungsproblem zu kodieren werden zwei Dateien benötigt. In einer Datei wird die Domäne beschrieben, d. h. welche Objekte, Eigenschaften, Funktionen und Aktionen es geben kann. In der anderen wird das tatsächliche Problem definiert. Dazu wird der Initialzustand, der Zielzustand, die tatsächlich existierenden Objekte und die Belegung der Prädikate und Funktionen hinterlegt. Durch diese Teilung kann die Domäne wieder verwendet werden, um verschiedene Probleme zu lösen.

Basierend auf diesen zwei Dateien führt ein automatischer Planer seinen Planungsprozess durch. Als Lösung liefert er eine Folge von Aktionen, auch Plan genannt.

PDDL enthält noch mehr Schlüsselworte, beispielsweise um mit probabilistischen Aktionen umzugehen. Zudem existieren zahlreiche Erweiterungen. In [7] findet sich eine detaillierte Beschreibung von PDDL.

2.4 Verwandte Ansätze

Mamei und Zambonelli entwickelten die TOTA Middleware [21]. In ihrer Arbeit konzentrieren sie sich auf Selbst-Organisation. TOTA unterstützt adaptive und entkoppelte Interaktionen zwischen Agenten. Dabei wird auf Tupeln basierende Kommunikation verwendet. Nachrichten haben Regeln, nach denen sie sich über das Netzwerk verteilen.

CARISMA [26] ist eine Service orientierte Middleware, welche Selbst-Konfiguration und Selbst-Optimierung unter Berücksichtigung von real-time Fähigkeit umsetzt. Basierend auf lokalen Informationen berechnet ein Service ob er einen Job ausführen kann und zu welcher Qualität. Die Middleware entscheidet mit Hilfe eines Auktion-Systems welcher Service den Job ausführen darf. Zudem startet und stoppt sie Service-Agenten.

Die Artificial Hormone System (AHS) Middleware [48] wurde für eingebettete Systeme entwickelt. Sie ist an das Hormonsystem höher entwickelter Säugetiere angelehnt. Hormone werden in Form von Nachrichten realisiert. Jedes processing element berechnet wie gut es geeignet ist eine Aufgabe auszuführen. Das Element mit dem höchsten Wert führt die Aufgabe aus. Falls mehr als eine Aufgabe verteilt werden muss wird die Prozedur wiederholt. Die Hauptaufgabe dieser Middleware besteht in der Verteilung von Aufgaben zu processing elements.

ORCA [22] ist eine Architektur um Roboter zu steuern. ORCA basiert auf einer hierarchischen Systemarchitektur mit vielen Kontrolleinheiten. Ziel ist es, das Verhalten des Systems zu verbessern und auf Fehler zu reagieren ohne ein formales Modell zu verwenden. ORCA verwendet überwachtes Lernen um die Selbst-Organisation des Systems zu kontrollieren.

OCμ konzentriert sich im Gegensatz zu CARISMA, ORCA und teilweise AHS auf allgemeine Systeme ohne real-time Fähigkeit. Zudem muss ein Service keine Selbst-X Eigenschaft selbst umsetzten oder der Middleware zusätzliche Funktionen, wie beispielsweise eine Einschätzung der erreichbaren Qualität zur Verfügung stellen. In AHS und TOTA werden viele zusätzliche Nachrichten kreiert, in OCμ wird versucht zusätzliche Netzwerklast gering zu halten. In CARISMA und AHS wird jede Aufgabe einzeln vergeben, die Verteilung von multiplen Aufgaben wird nicht betrachtet.

Seebach [42] beschäftigt sich mit der Konstruktion selbst-organisierender Software-Systeme. Als Fallbeispiel verwendet sie eine Produktionsanlage mit mehreren Maschinen. Jede Maschine kann aus einer Menge von verschiedenen Werkzeugen auswählen. Zwischen den Maschinen fahren Carts die Werkstücke hin und her. Wenn ein Werkzeug defekt ist oder ein Cart ausfällt, kann sich das System neu konfigurieren. Dazu wird ein verteilter Algorithmus verwendet, der auf Koalitionsbil-

dung basiert. Im Gegensatz zu der vorliegenden Arbeit ist eine Rekonfiguration des Systems aufwändig und die verschiedenen Teilsysteme müssen dabei gleichzeitig neu konfiguriert werden. Zudem kann eine Produktionsanlage nicht weiterlaufen, während die Rekonfiguration erfolgt.

Innerhalb des CAR-SoC Projekts [20], [19] wird eine zweistufiges Verfahren verwendet um Hardware und Software Komponenten einer eingebetteten Kontrolleinheit in einem Auto zu kontrollieren. Jede Komponente nützt einen eventuell unvollständigen MAPE Zyklus mit einfachen Regeln in der Planungsphase. Aufsetzend auf diesen Modulen steuert ein globaler MAPE Zyklus das System mit Hilfe von Klassifikatoren. Es kann keine Kette von Aktionen erzeugt oder gelernt werden.

Im Projekt „Organic Control of Traffic Lights“ [33] wird ebenfalls ein zweistufiger Ansatz verwendet um Ampeln zu steuern. Ein Learning Classifier System (LCS) [10] wählt Aktionen zur Laufzeit aus, während ein evolutionärer Algorithmus offline auf einigen leistungsstarken Knoten neue Classifier erzeugt und in einer simulierten Umgebung evaluiert.

Der in dieser Arbeit vorgestellte Reflex Manager kann in Zusammenarbeit mit dem Planner Manager ebenfalls als zweistufiges Planungssystem gesehen werden. Er arbeitet jedoch online auf dem tatsächlichen System. Zudem kann ein LCS in jedem Planungsschritt nur eine einzige Aktionen auswählen und ausführen, während der hier verwendete Planer und Reflex Manager Pläne erstellen und verwalten, welche aus mehreren Aktionen bestehen.

Der Reflex Manager verwendet Pläne wieder, die ursprünglich für einen anderen, ähnlichen Zustand erstellt wurden. Er verändert diese Pläne jedoch nicht. Eine Manipulation der Pläne ist laut [25] mindestens genauso aufwändig, wie die Erstellung eines neuen Planes. Deshalb wurde in dieser Arbeit von einer Anpassung der Pläne abgesehen.

In OCμ wird als automatischer Planer JavaFF [3] verwendet, der auf dem FF-Planer [9] basiert. Er erstellt einen sequentiellen, korrekten Plan. Ein SAT-Solver [8] kann ebenfalls als Werkzeug der Entscheidungsfindung dienen. Er erreicht nur eine gültige Lösung, nicht jedoch eine optimale. Seine Planungszeit ist geringer, aber es entstehen durch eine nicht optimale Lösung Mehrkosten. Dasselbe Problem ergibt sich bei genetischen Algorithmen. Sie finden ein Lösung, die jedoch nicht optimal sein muss. Zudem ist die Angabe einer Zielfunktion notwendig, die optimiert wird.

3 OCμ Architektur

In diesem Kapitel wird die grundlegende Architektur von OCμ erklärt. OCμ ist eine Organic Computing Middleware und abstrahiert von der Netzwerkprogrammierung. So können unterschiedliche Netzwerke verwendet werden, ohne dass eine Anwendung darauf Rücksicht nehmen muss.

Zudem unterstützt sie Interoperabilität. Sie ist auf verschiedenen Plattformen und Betriebssystemen verfügbar. Des Weiteren verfügt OCμ über gewisse Basisdienste, die in einem verteilten System immer wieder benötigt werden. Beispielsweise ermöglicht sie das Registrieren und Finden von Diensten.

3.1 Geschichte

Abbildung 3.1 zeigt die ursprüngliche Architektur eines OCμ Knotens [43]. Diese Architektur ist auf jedem OCμ Knoten implementiert. OCμ ist eine nachrichten-

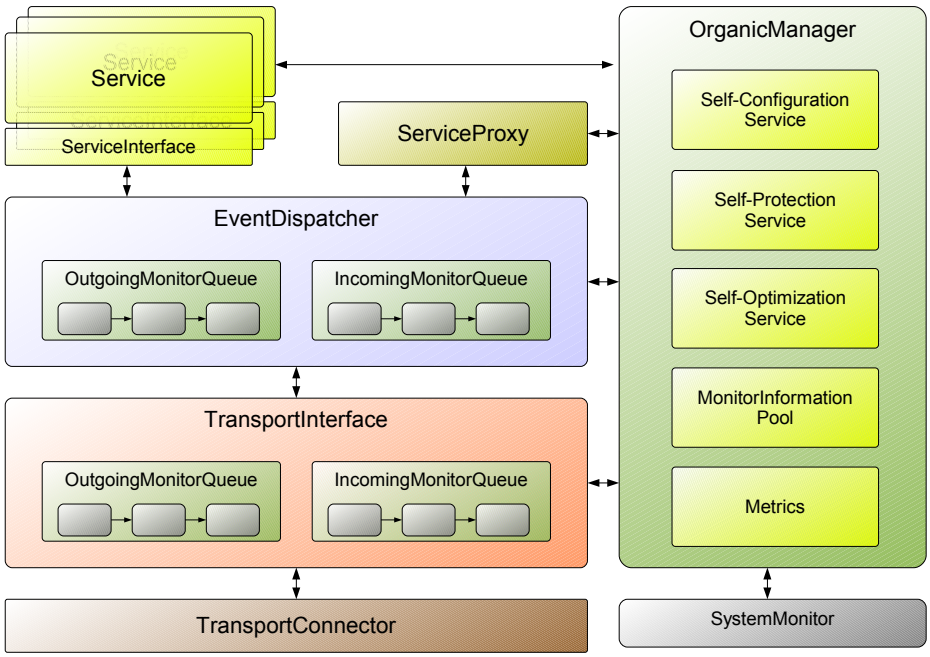


Abbildung 3.1: Ursprüngliche Architektur eines OCμ Knotens

basiert Middleware. Der *TransportConnector* ist für die Abstraktion der Netzwerkschicht zuständig während der *EventDispatcher* die Nachrichten verteilt. Beide verfügen über *MonitorQueues*, die von den Nachrichten durchlaufen werden. Services werden über ein *ServiceInterface* angebunden.

Der größte Unterschied zur aktuellen Architektur zeigt sich im Organic Manager. Im ursprünglichen Organic Manager gibt es je einen Service für jede Selbst-X Eigenschaft.

Der Algorithmus zur *Selbst-Konfiguration* [43], [45] basiert auf sozialem kooperativem Verhalten. Jeder Service muss angeben, welche Ressourcen er benötigt. Der Selbst-Konfiguration-Service berechnet, mit welcher Dienstgüte er den Service ausführen kann. Mit Hilfe eines verteilten Algorithmus wird festgestellt, welcher Knoten am besten geeignet ist und dort wird dieser eine Service gestartet. Dieses Verfahren muss für jede Service-Instanz durchgeführt werden.

Die *Selbst-Optimierung* [43], [47] orientiert sich am menschlichen Hormonsystem. Abhängig von der Last der Knoten berechnet der Selbst-Optimierungs-Service Hormonwerte und verbreitet diese mit Hilfe von Nachrichten. Diese Methode führt zu einem Lastausgleich zwischen den Knoten.

Techniken des biologischen Immunsystems wurden zur Umsetzung des *Selbst-Schutzes* [29], [28] verwendet. Dabei analysiert und bewertet der Selbst-Schutz-Service Nachrichten zwischen Knoten um sie als harmlos oder gefährlich zu klassifizieren.

Für die *Selbst-Heilung* [35] wurde ein Failure Detector Service implementiert um zu erkennen, ob ein Knoten ausgefallen ist. Er kann bestehende Nachrichten verwenden. Bei hohem Nachrichtenaufkommen belastet er das Netz nicht zusätzlich.

Jeder dieser Services wurde unabhängig voneinander entwickelt und sammelt die benötigten Daten selbständig. Wechselwirkungen zwischen den Services wurden nicht beachtet.

Diese Selbst-X Eigenschaften haben jedoch zahlreiche Gemeinsamkeiten. Beispielsweise basieren die Selbst-Konfiguration, Selbst-Optimierung und Selbst-Heilung auf Informationen über die im Netz laufenden Knoten und ihre Services. Zudem befassen sich die Selbst-Konfiguration und die Selbst-Optimierung damit, welcher Ort der geeignetste für einen Service ist. Da sie verschiedene Verfahren verwenden ist es nicht ausgeschlossen, dass die Selbst-Konfiguration einen Knoten wählt und einen Service startet und die Selbst-Optimierung ihn anschließend verschiebt.

Um die Zusammenhänge zwischen den Selbst-X Eigenschaften und die Erfahrungen aus der Middleware berücksichtigen zu können, wurde die Architektur wie im nächsten Abschnitt beschrieben, weiterentwickelt. Ziel war es unter anderem, eine gemeinsame Datenbasis für die Selbst-X Eigenschaften zu schaffen. Außerdem

wurden die Selbst-Konfiguration, Selbst-Optimierung und Selbst-Heilung durch ein einziges Verfahren - einem automatischen Planer - umgesetzt, um Wechselwirkungen besser berücksichtigen zu können.

3.2 Aktuelle Architektur

Abbildung 3.2 zeigt die aktuelle Architektur von OCμ. Die klassischen Aufgaben einer Middleware werden von der linken Seite erfüllt. Der Organic Manager auf der rechten Seite überwacht die Middleware und greift steuernd ein, falls notwendig. Für den Organic Manager ist die Middleware ein *System under observation and control* (SuOC).

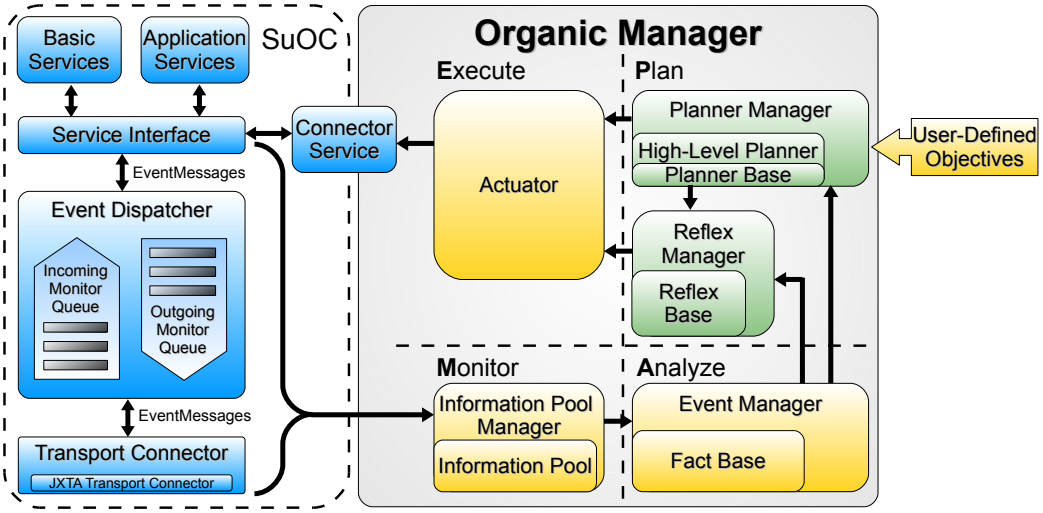


Abbildung 3.2: Architektur eines OCμ Knotens

Anwendungen werden als Services implementiert und können auf mehrere Knoten verteilt werden. Die Kommunikation zwischen Knoten findet ausschließlich über *EventMessages* statt. Services können mit Hilfe des Service Interfaces Nachrichten generieren und versenden. Der Event Dispatcher verteilt eingehende Nachrichten an Services und nimmt ausgehende Nachrichten von Services entgegen. Er verwaltet zudem die eingehende und ausgehende Monitor Queues. Nachrichten von Services durchlaufen erst die komplette ausgehende Monitor Queue bevor sie an den Transport Connector weitergeleitet werden. Monitore können unter anderem Informationen an ausgehende Nachrichten per piggy-back anhängen.

Nachrichten vom Transport Connector werden erst von allen Monitoren der eingehenden Monitor Queue abgearbeitet. Dabei können piggy-back Informationen wieder von der Nachricht entfernt werden. Anschließend wird die Nachricht zugestellt.

Der Transport Connector ist für die tatsächliche Datenübertragung zwischen einzelnen Knoten zuständig. Er ist als Interface gestaltet und kann für verschiedene Übertragungsmedien implementiert werden. Um Nachrichten zwischen PCs zu verschicken kann der JXTA Transport Connector [2] genutzt werden. Im Laufe des OCμ Projekts wurde ein Transport Connector entwickelt, der es ermöglicht mehrere OCμ Knoten auf einem lokalen Rechner laufen zu lassen. Des Weiteren wurde ein Socket Transport Connector implementiert, der ein internes Netzwerk nutzen kann und mit Hilfe von sockets kommuniziert.

Eine Besonderheit von OCμ besteht in der Möglichkeit Services als Ganzes zu verschieben. Wenn ein Service verschoben werden soll, kann er der Middleware seine Daten übergeben und wird anschließend beendet. Dann wird der komplette Services auf einen anderen Knoten übertragen und neu gestartet. Dabei erhält er seine von ihm gespeicherten Daten und kann sofort weiter arbeiten. Auf dem ursprünglichen Knoten wird ein Service Proxy gestartet. Er leitet Nachrichten, die für den verschobenen Service bestimmt waren, an den neuen Standort weiter. Nach einer gewissen Zeit kann der Proxy Service beendet werden.

Der Organic Manager dient dazu die Middleware zu überwachen und zu verbessern. Er orientiert sich am MAPE (Monitor, Analyze, Plan, Execute) Zyklus. In der Monitor Phase werden Daten gesammelt und gespeichert. Anschließend erfolgt eine Analyse um belastbare Daten zu erhalten. Basierend auf dieser Datengrundlage werden in der Planungsphase Aktionen gewählt um das System zu verbessern. In der Execute Phase werden diese Aktionen ausgeführt. In den folgenden Abschnitten wird auf jede Phase genauer eingegangen.

3.2.1 Monitoring Phase

Die verfügbaren Informationen können in zwei Klassen eingeteilt werden: Lokale Informationen über den eigenen Knoten und externe Informationen über andere Knoten. Die CPU und Speicherplatzauslastung sowie die lokal aktiven Services sind lokale Informationen, die direkt ermittelt und an den Information Pool Manager geleitet werden können. Externe Informationen werden als piggy-back auf Nachrichten empfangen und auch an den Information Pool Manager geleitet. Des Weiteren wird bei der Verschiebung eines Services der Information Pool Manager benachrichtigt. Der Information Pool Manager entscheidet ob die Information in den Information Pool gespeichert wird. Wenn die bereits gespeicherten Daten neuer sind als die ihm angebotenen verändert er den Information Pool nicht. Falls nötig informiert er den Event Manager über Änderungen in den Daten.

3.2.2 Analyze Phase

Hauptaufgabe der Analyze Phase ist es, die gesammelten Informationen zu aggregieren und als Facts abzuspeichern. Der Event Manager zählt beispielsweise Services gleichen Typs und speichert nur noch ihre Anzahl ab. Diese kompaktere Datenbasis reduziert die notwendige Planungskomplexität in der nächsten Phase. Die aggregierten Informationen werden regelmäßig an den Planner Manager weiter geleitet.

3.2.3 Planning Phase

Nachdem der Planner Manager angestoßen wurde, kontrolliert er ob sich das System im gewünschten Zustand befindet oder ob ein Eingreifen nötig ist. In diesem Fall übersetzt er das Problem in PDDL, eine Sprache die für neuere automatische Planer Standard ist. Die dafür benötigten Information entstammten der Fact Base, der Planner Base und den User-Defined Objectives. In der Planner Base sind beispielsweise Informationen über Ziele und Aktionen gespeichert. Die Ziele der Anwender finden sich in den User-Defined Objectives.

Anschließend stößt der Planner Manager den automatischen Planer an. Dieser erstellt einen Plan der vom aktuellen Zustand zum Zielzustand führt. Der Planner Manager leitet diesen Plan an den Actuator und den Reflex Manager weiter. Der Reflex Manager speichert Pläne um sie wieder zu verwenden und die Reaktionszeit des System zu verkürzen. Er wird in Abschnitt 4.2 näher betrachtet. Eine genauere Beschreibung des Planer und seiner Modelle befindet sich in Abschnitt 4.1.

3.2.4 Executing Phase

Der Actuator erhält Pläne vom Planner Manager und vom Reflex Manager. Jeder Plan besteht aus einer Liste von Aktionen. Er führt die Aktionen hintereinander aus, wobei eine Aktion durchaus entkoppelt sein kann und noch ausgeführt wird während der Actuator die nächste aufruft. Falls der Reflex Manager und der Planner Manager unterschiedliche Pläne liefern, kann der Actuator wie in Kapitel 4.3 beschrieben reagieren.

Mit Hilfe der User-Defined Objectives kann ein Anwender dem automatischem Planer zusätzliche Ziele vorgeben. Im Augenblick kann der Anwender entscheiden welcher Service wie oft im Netz laufen soll. Zudem wurde ein Simulator entwickelt in dem die User-Defined Objectives mit Hilfe eines graphischen Interfaces gesetzt werden können. Diese werden dann von dem lokalen Planer umgesetzt. Auch Services haben über das Service Interface die Möglichkeit dem Planer Ziele, sowie zusätzliche Aktionen zu übergeben.

4 Organic Manager - Planning und Execute Phase

Auf jedem Knoten überwacht und steuert ein Organic Manager die Middleware. Wie bereits in Kapitel 3 beschrieben ist er als MAPE-Zyklus implementiert. Herzstück des Organic Manager ist der Planner Manager. Er entscheidet auf Grundlage der vorliegenden Informationen wie das System beeinflusst wird. In OCμ übernimmt ein automatischer Planer diese Aufgabe.

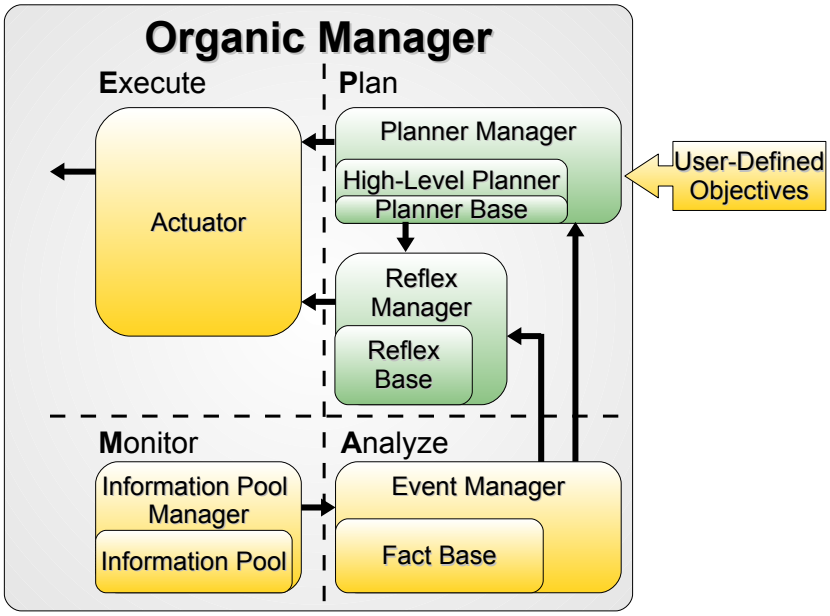


Abbildung 4.1: Organic Manager

Im folgenden Abschnitt werden die Aufgaben und die Funktionsweise des automatischen Planers näher erläutert. In Abschnitt 4.2 wird das Konzept eines Reflex Managers vorgestellt. Er speichert Pläne um sie später wieder verwenden zu können. Die Ausführung eines Planes wird vom Actuator übernommen. Abschnitt 4.3 beschreibt die Funktionsweise des Actuators und wie er mit konkurrierenden Plänen des Planner Manager und des Reflex Managers umgeht.

4.1 Automatischer Planer

Der automatische Planer setzt die drei Selbst-X Eigenschaften *Selbst-Konfiguration, Selbst-Optimierung* und *Selbst-Heilung* um. In diesem Kapitel werden diese Eigenschaften konkretisiert und die Anbindung des automatischen Planers JavaFF erläutert. Zudem werden zwei Modelle vorgestellt, die die Selbst-X Eigenschaften umsetzen. Abschließend werden diese Modelle in einer Evaluation miteinander verglichen. Die PDDL Modelle und ihre Evaluierungsergebnisse wurden in Artikel [41] veröffentlicht.

Im Folgenden wird jede dieser Eigenschaften allgemein beschrieben, angelehnt an die Definition in [13] und wie diese Eigenschaften in OCμ umgesetzt werden.

Selbst-Konfiguration Das System konfiguriert sich selbst automatisch und folgt dabei höheren Zielen. Eine neue Komponente fügt sich z. B. nahtlos in das bestehende System ein. In OCμ ist eine Komponente ein OCμ Knoten. Ein OCμ Knoten startet alle nötigen Basisdienste und zusätzlich nötige Services selbständig. Zusätzliche Services können vom Anwender oder von Services definiert werden. Der Organic Manager erhält diese dann als weitere Ziele und erfüllt sie, beispielsweise indem er die nötigen Services startet.

Selbst-Optimierung Während der Laufzeit versuchen die Komponenten ständig ihre eigene Leistung und Effizienz zu verbessern. In OCμ wird dieses Ziel durch eine gleichmäßige Auslastung aller Knoten realisiert. Falls ein Knoten bei sich eine überdurchschnittlicher Auslastung feststellt, verschiebt er Services auf weniger ausgelastete Knoten. Dies führt zu einem Lastausgleich.

Selbst-Heilung Das System entdeckt, diagnostiziert und behebt Fehler selbstständig. In OCμ kann der Komplettausfall eines Knotens erkannt werden. Die Organic Manager der anderen Knoten analysieren, welche Services durch diesen Ausfall fehlen. Dann führen sie das System wieder in einen validen Zustand, indem sie die entsprechenden Services neu starten.

Für die Erfüllung dieser drei Selbst-X Eigenschaften ist der Organic Manager verantwortlich. Er verwendet einen automatischen Planer der entscheidet, welche Aktionen ausgeführt werden.

Die ursprünglichen Selbst-X Eigenschaften verwenden teilweise die gleichen Aktionen. Beispielsweise nutzen die Selbst-Konfiguration und die Selbst-Heilung die Möglichkeit Services starten zu können. Zudem beeinflussen sich die Selbst-X Eigenschaften gegenseitig. So kann die Selbst-Heilung durch das Starten eines Services eine Selbst-Optimierung nach sich ziehen. Oder die Selbst-Optimierung ist nicht mehr notwendig, weil die Selbst-Konfiguration einen Service gestoppt hat.

Um diese Effekte besser berücksichtigen zu können, wird ein automatischer Planer verwendet, der alle drei Selbst-X Eigenschaften zusammen umsetzt.

4.1.1 Anbindung des Automatischen Planers

Der Planner Manager stellt das Framework für einen automatischen Planer dar. Um einen Plan erstellen zu können, werden verschiedene Informationen benötigt, wie in Abschnitt 2.3.1 beschrieben.

Das tatsächliche Planungsproblem wird bei jedem Aufruf des Planers neu erstellt. Unter anderem enthält es den Ausgangszustand des Systems, also die aktuellen beobachteten Werte. Diese Werte werden von der Fact Base an den Planner Manager gesendet. Diese Informationen werden unabhängig vom internen Modell des automatischen Planers gesammelt und gespeichert. Bevor der automatische Planer diese Informationen verarbeiten kann, müssen sie in PDDL übersetzt werden. Diese Übersetzung geschieht abhängig vom verwendeten Modell.

Der Zielzustand ist in der Planner Base gespeichert und kann zur Laufzeit geändert werden. Er setzt sich aus einzelnen Zielen zusammen, die in PDDL übersetzt werden. In ihm sind auch die Ziele der Anwender enthalten.

Die einzelnen Aktionen werden ebenfalls in der Planner Base gespeichert. Zusammen mit den grundsätzlich möglichen Prädikaten, Funktionen und Objekten aus der Planner Base stellen sie die Domäne des Planers dar. Diese Informationen werden in PDDL übersetzt und als eine Textdatei gespeichert. Die Domäne ist im Regelfall statisch und wird für jedes Planungsproblem wiederverwendet. Wenn jedoch z. B. neue Aktionen der Planner Base hinzugefügt werden, muss die Domäne neu erstellt werden. Dabei werden die Informationen wieder im PDDL Format abgespeichert und die alte Datei wird überschrieben.

Der automatische Planer erhält als Eingabe die Domäne und ein Problem. Darauf basierend liefert er einen Plan zurück. In OCμ wird als automatischer Planer JavaFF [3] verwendet. Er basiert auf den FFPlaner von Hoffmann und Nebel [9]. Ein von JavaFF erstellter Plan besteht aus einer geordneten Abfolge von Aktionen mit ihren Parametern. Eine Aktion wird eindeutig durch ihren Namen identifiziert. Durch diesen Namen kann der Actuator die Aktion finden und aufrufen, wobei er der Aktion die vom Planer gewählten Parameter übergibt.

4.1.2 Boolesches Modell

Das erste Modell verwendet ausschließlich Boolesche Prädikate und keine Funktionen. Als möglicher Typ von Objekten wird unter anderem der lokale Knoten verwendet. Er kommt im Planungsproblem nur ein einziges Mal vor. Als weiterer Typ wird ein Service verwendet:

```
(:types
service - object
HomeNode - object)
```

Da es verschiedene Services gibt, kann dieser Typ im Planungsproblem öfters verwendet werden. Die Anzahl an Objekten des Typs `service` bestimmt sich aus der Anzahl unterschiedlicher Services. Jeder Service verfügt über je ein Prädikat, das zeigt, ob zu viele oder zu wenig Instanzen dieses Services laufen. Dies wird mit Hilfe der User-Defined Objectives ermittelt. Ein weiteres Prädikat gibt an, ob zumindest eine Instanz des Services lokal läuft, da nur in diesem Fall der Service verschoben werden kann. Der lokale Knoten verfügt über ein Prädikat, das angibt, ob er überlastet ist. Ein Knoten ist überlastet, wenn seine Auslastung höher ist als die durchschnittliche Auslastung aller ihm bekannten Knoten.

Das Boolesche Modell verfügt über drei Aktionen. Ein Service kann gestartet werden, wenn das Prädikat, welche anzeigt, dass zu wenige Instanzen dieses Services vorhanden sind, wahr ist. Als Effekt wird der Wert dieses Prädikates auf falsch geändert. Zudem ist der Service anschließend lokal verfügbar und kann somit auch verschoben werden. Falls zu viele Instanzen eines Services laufen und dieses Prädikat wahr ist, kann ein Service gestoppt werden. Danach wird dem Prädikat der Wert falsch zugewiesen:

```
(:action stop
:parameters (?x - service)
:precondition (tooManyInstances ?x)
:effect (not (tooManyInstances ?x))
)
```

Falls bei der Ausführung der Aktion `Stopp` festgestellt wird, dass der Service nicht lokal läuft, wird er auf einem anderen Knoten gestoppt. Als letzte Aktion kann ein Service verschoben werden, wenn er lokal läuft, verschiebbar ist und der lokale Knoten überlastet ist. Als Effekt werden diese drei Prädikate auf falsch gesetzt.

Die offensichtliche Schwäche dieses Ansatzes besteht in der Unfähigkeit, mehrere z.B. lokal laufende Instanzen eines Services in einem Planungsschritt zu beeinflussen. Wenn zwei Instanzen eines Services laufen sollen und es läuft noch keine, muss der Planer zweimal angestoßen werden und startet dann je eine Instanz. Die Stärke des Ansatzes liegt jedoch in seiner Einfachheit. Die Komplexität des Planungsproblems ist gering, was sich in einem kurzen und wenig verzweigtem Suchbaum zeigt.

4.1.3 Numerisches Modell

Das zweite PDDL Modell verwendet Funktionen und kann somit numerische Werte als Eigenschaften von Objekten verwenden. Neben dem lokalen Knoten und den Services gibt es noch ein Objekt, welches das Netzwerk darstellt:

```
(:types
Service - object
HomeNode - object
Network - object
)
```

Im Planungsproblem gibt es jeweils nur einen lokalen Knoten und ein Objekt, welches das Netzwerk repräsentiert. Die Anzahl an Objekten des Typs `service` wird von der Anzahl unterschiedlicher Services bestimmt.

Numerische Werte werden dem PDDL Modell als Funktionen (`functions`) übergeben. Listing 4.1 zeigt die verwendeten Funktionen. Die Werte von `upperTarget` und `lowerTarget` legen fest, wie viele Instanzen eines Services laufen sollen. Mit Hilfe numerischer Werte wird auch beschrieben, wie viele Service-Instanzen tatsächlich laufen und wie viele lokal vorhanden sind. Die Anzahl verschiebbarer Services des lokalen Knotens sowie seine Last sind auch als PDDL Funktionen umgesetzt. Das Netzwerk besitzt einen numerischen Wert, der die durchschnittliche Auslastung aller Knoten angibt. Zusätzlich verfügt es über den Wert `netScale`, der angibt, wie stark sich diese durchschnittliche Auslastung ändert, wenn ein Service auf einem anderen Knoten gestartet, gestoppt oder dorthin verschoben wird.

```
(:functions
(upperTarget ?ser - Service)
(lowerTarget ?ser - Service)
(overallRunning ?ser - Service)
(localRunning ?ser - Service)
(relocatableServices ?hom0 - HomeNode)
(myLoad ?hom - HomeNode)
(netLoad ?net - Network)
(netScale ?net - Network)
)
```

Listing 4.1: Funktionen im Numerischen Modell

Das Numerische Modell kann ebenfalls Services starten, stoppen und verschieben. Es verfügt insgesamt über zehn Aktionen, sieben mehr als das Boolesche Modell. Es gibt drei Aktionen um einen Service zu starten. Algorithmus 1 zeigt die

Unterschiede. Die ersten zwei Aktionen starten auf dem lokalen Knoten einen verschiebbaren bzw. nicht verschiebbaren Service. Die dritte Aktion startet einen Service auf einem anderen Knoten im Netzwerk.

Um einen Service starten zu können, müssen grundsätzlich weniger Instanzen als nötig laufen, als Effekt wird die Anzahl insgesamt laufender Instanzen dieses Services um eins erhöht. Die erste `start` Aktion startet lokal einen verschiebbaren Service und erhöht als zusätzlichen Effekt die Anzahl verschiebbarer Services, die Anzahl lokal laufender Services und die Auslastung. Die zweite Aktion startet lokal einen nicht verschiebbaren Service. Im Gegensatz zur vorherigen Aktion wird die Anzahl verschiebbarer Services nicht verändert. Die dritte Aktion startet eine Instanz des Services auf einen Knoten im Netzwerk und erhöht die durchschnittliche Auslastung des Netzwerks. Analog ist in drei Aktionen das Stoppen eines Services modelliert.

Algorithmus 1 Numeric Modell: Start Services

Require: overallRunning<lowerTarget
1: **if** relocatable **then**
2: **do action:** StartRelocatableService
3: relocatableServices = + 1
4: myLoad = +1
5: localRunning = + 1
6: **end if**
7: **if** not relocatable **then**
8: **do action:** startNonRelocatableService
9: myLoad = + 1
10: localRunning = + 1
11: **end if**
12: **if** myLoad > netLoad **then**
13: **do action:** startServiceElsewhere
14: netLoad = + netScale
15: **end if**
16: overallRunning = + 1

Eine weitere Aktion ermöglicht es, einen Service zu verschieben. Voraussetzung ist, dass die Auslastung des lokalen Knotens höher ist als die durchschnittliche Auslastung, ein Service verschiebbar ist und die Anzahl lokal laufender Services positiv ist. Als Effekt verringert sich die Anzahl lokal laufender Instanzen des Services sowie die Auslastung des lokalen Knotens und die Anzahl verschiebbarer Services der lokalen Knotens, wohingegen sich die durchschnittliche Auslastung erhöht. In PDDL lässt sich die Aktion wie folgt darstellen:

```
(:action relocateService
:parameters (?x - Service ?h - HomeNode ?n - Network)
```

```
:precondition (and  (> (myLoad ?h) (netLoad ?n) )
                    (> (localRunning ?x) 1)
                    (relocatable ?x) )
:effect (and  (decrease (localRunning ?x) 1 )
              (decrease (myLoad ?h) 1)
              (decrease (relocatableServices?h) 1)
              (increase (netLoad ?n) (netScale ?n)))
)
```

Da numerische Werte nicht als Ziele von JavaFF gesetzt werden können, gibt es noch zwei weitere Prädikate und drei Aktionen um numerische Werte in Boolesche Werte umzuwandeln. Die drei Aktionen enthalten als Vorbedingung numerische Werte und als Effekt einen Booleschen Wert, der als Ziel verwendet werden kann. Die Aktionen dienen dazu, folgende Situationen bzw. Ziele abzudecken:

- Ziel gleichmäßige Auslastung: Auslastung des lokalen Knotens ist ähnlich der durchschnittlichen Auslastung
- Gleichmäßige Auslastung nicht erreichbar weil z. B. zu wenig verschiebbare Services lokal vorhanden sind
- Ziel Anzahl Service-Instanzen: Vorgegebene Anzahl an Service-Instanzen ist erreicht

Eine Aktion setzt ein Prädikat auf wahr, wenn die Auslastung des lokalen Knotens der durchschnittlichen Auslastung nahe genug ist. Der Planer erhält als Ziel nur, dass dieses Prädikat wahr sein soll. Eine andere Aktion kann verwendet werden, wenn die Anzahl verschiebbarer Services nicht ausreicht um diesen Zustand zu erreichen, beispielsweise weil auf dem lokalen Knoten zu viele nicht verschiebbare Services laufen. Diese Aktion führt dazu, dass der Planer nicht abbricht, wenn er keine gleichmäßige Lastverteilung erreichen kann. Somit kann der Planer immer noch einen Plan erstellen, welcher andere, erreichbare Ziele erfüllt. Ob die gewünschte Anzahl an Instanzen für jeden Service erreicht wurde, kontrolliert eine weitere Aktion.

Die Stärke dieses Modells liegt in seiner Fähigkeit mehrere Instanzen in einem Planungsschritt starten, stoppen oder verschieben zu können. Sein Nachteil liegt jedoch in der höheren Planungskomplexität durch einen breiteren Suchbaum, bedingt durch die größere Anzahl an möglichen Aktionen. Zudem ist die Tiefe des Suchbaums durch die Möglichkeit, mehrere Instanzen zu verwalten, höher als beim Booleschen Modell.

4.1.4 Evaluierung

Der Organic Manager setzt Selbst-Optimierung, Selbst- Konfiguration und Selbst-Heilung um. Sowohl das Boolesche, als auch das Numerische Modell können diese Eigenschaften realisieren. Im Folgenden werden die einzelnen Selbst-X Eigenschaften evaluiert.

Evaluierungsbasis

Jeder Knoten verfügt über einige elementare Services. Diese Services sind auf jedem Knoten identisch. Die Auslastung eines Knoten wird durch die zusätzliche Anzahl an Services bestimmt, die ergänzend zu den elementaren Services laufen. In den folgenden Evaluierungen wird der automatische Planer sekündlich angestoßen. Nach Ausführung des Planes bleibt somit noch Zeit, um die Veränderungen des Systems zu beobachten und im Information Pool abzuspeichern. Dadurch enthält die Fact Base alle relevanten Änderungen, bevor der Planer das nächste Mal angestoßen wird.

Ein System aus mehreren Knoten ist balanciert, wenn folgende Definition zutrifft:

Definition 1 *Sei Knoten j derjenige mit der geringsten Auslastung l_j in einem System S, dann ist S balanciert, wenn jeder andere Knoten i entweder die gleiche Anzahl an Services l_i oder maximal einen Service mehr hat:*

$$\exists j : \forall i : l_j \leq l_i \leq l_j + 1 \; mit \; i, j \in S$$

Szenarien

Das System befindet sich in einem stabilen Zustand, wenn kein Service mehr verschoben wird. Um die zwei entwickelten Modelle zu evaluieren, wird ein Netz aus mehreren Knoten verwendet, wobei auf jedem Knoten ein Organic Manager mit automatischem Planer läuft. Alle Knoten laufen auf einem PC. Sie unterscheiden sich von verteilt laufenden OCμ Knoten lediglich durch den verwendeten Transport Connector. Im Folgenden werden vier Szenarien betrachtet:

- *Selbst-Optimierung*: Services sollen gleichmäßig im Netz verteilt werden
- *Selbst-Konfiguration*: Es soll eine vorgegebene Anzahl an Services gestartet und gleichmäßig verteilt werden
- *Selbst-Heilung*: Nach Ausfall von Knoten sollen die fehlenden Services gestartet und im Netz verteilt werden

- *Eintritt eines neuen Knotens*: Ein neuer Knoten tritt einem balanciertem und stabilem Netz bei

Die Anzahl verwendeter Knoten hat nur einen geringen Einfluss auf das Verhalten der Planers, da die Planungsmodelle keine Information über jeden einzelnen Knoten enthalten. Deshalb wird in den folgenden Evaluierungen auf eine Variation der Knotenanzahl verzichtet. Eine ausführlichere Evaluation findet sich in Kapitel 5.

Selbst-Optimierung

Dieses Szenario zeigt, dass beide Modelle die Selbst-X Eigenschaft Selbst-Optimierung umsetzen. Aufgabe des Planers ist es, Services möglichst gleichmäßig zu verteilen.

In diesem Szenario wird ein Netz mit 10 Knoten zu Grunde gelegt. Jeder Knoten verfügt über einen automatischen Planer mit dem Ziel Selbst-Optimierung. Auf einem Knoten werden n zusätzliche Services gestartet, wobei $n \in (2, \ldots, 25)$ ist.

Abbildung 4.2 zeigt die Zeit, die nötig ist, bis das System selbständig einen balancierten und stabilen Zustand erreicht hat. Auf der X-Achse sind die zusätzlich gestarteten Services aufgetragen. Bei zwei Services benötigen beide Modelle die

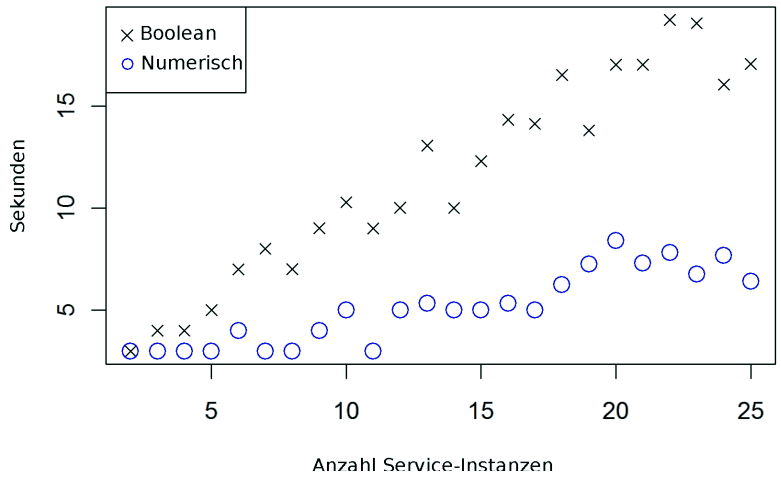


Abbildung 4.2: Dauer bis ein System aus 10 Knoten einen balancierten und stabilen Zustand erreicht hat

gleiche Zeit, da sie nur einmal planen müssen. Je mehr Services gestartet werden, desto länger braucht das Boolesche Modell, wohingegen sich das Numerische Modell relativ stabil verhält. Wenn z. B. 25 Services gestartet werden benötigt das Boolesche Modell 17 Sekunden, wohingegen das Numerische Modell nach 6 Sekunden

bereits einen balancierten und stabilen Zustand erreicht hat. Wie bereits beschrieben liegt die Ursache für dieses Verhalten in der Notwendigkeit des Booleschen Modells mehrfach planen zu müssen, bis der gewünschte Zustand erreicht ist.

Beitritt eines neuen Knotens in ein bestehendes Netz

Wenn ein neuer Knoten einem bestehenden, balancierten System beitritt, verschieben andere Knoten Services auf ihn, da er eine geringere Auslastung hat. Ziel dieses Szenarios ist es, zu zeigen, dass der Beitritt eines neuen Knotens keine starken Fluktuation der Auslastung zur Folge hat. Stattdessen überführen beide Modelle das System in einen balancierten und stabilen Zustand.

Zu Beginn dieses Szenarios laufen zehn Knoten mit $n \in (11, \ldots, 25)$ verteilt laufenden Services. Das System befindet sich einem stabilen und balancierten Zustand. Ein elfter Knoten ohne Last tritt dem System bei.

In Abbildung 4.3 ist auf der y-Achse die Zeit aufgetragen, die das System benötigt um wieder einen stabilen und balancierten Zustand zu erreichen. Das Numerische Modell benötigt meist weniger Zeit, um sich an die neue Situation anzupassen als das Boolesche Modell. Wie bereits im ersten Szenario liegt die Ursache in der höheren Anzahl benötigter Planungsrunden des Booleschen Modells.

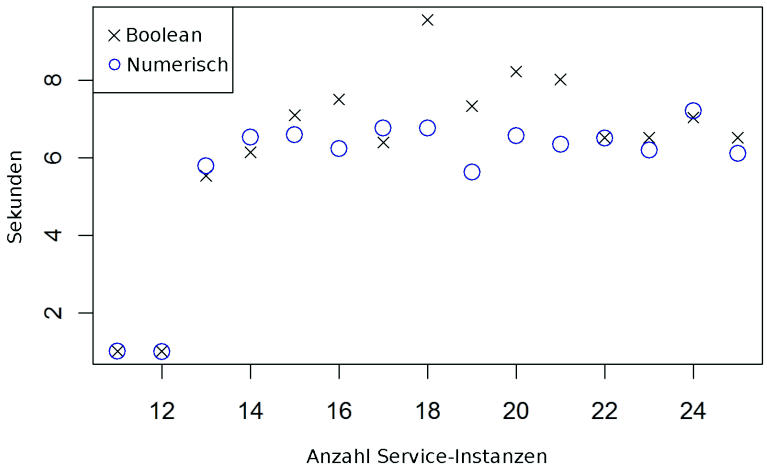


Abbildung 4.3: Benötigte Zeit bis wieder ein balancierter und stabiler Zustand erreicht wird, nachdem ein neuer Knoten einem bestehendem Netz beigetreten ist

Im Folgenden wird ein System mit 60 verteilten Services betrachtet. Abbildung 4.4 zeigt die Auslastung des neuen Knotens für die beiden Modelle. Beide Modelle erreichen einen balancierten und stabilen Zustand. Die neuen Knoten erhalten eine Auslastung von 6 Services. Dies ist die optimale Auslastung, da 60 Services über 11 Knoten verteilt eine Höchstzahl von 6 Services pro Knoten ergibt.

Hier zeigt sich ein kleiner Vorteil des Booleschen Modells. Da immer nur eine Instanz eines Services pro Planungsrunde verschoben werden kann, können in einer Planungsrunde maximal 10 Instanzen auf den neuen Knoten verschoben werden, von 10 Knoten je eine Instanz. Im Numerischen Modell können mehrere Instanzen verschoben werden. Dadurch steigt die Auslastung des neuen Knotens im Numerischen Modell stärker und höher als im Booleschen Modell. Auch mit dem Booleschen Modell erhält der neue Knoten zu viele Services. In beiden Modellen wird dies erkannt und durch den Planer in den nächsten Planungsrunden korrigiert. Obwohl das Numerische Modell mehr Services verschiebt, erreicht es schneller wieder einen balancierten und stabilen Zustand als das Boolesche Modell.

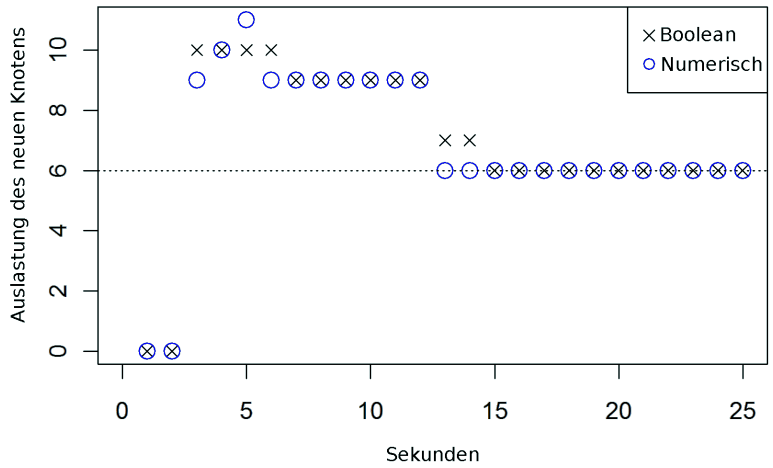


Abbildung 4.4: Auslastung des neuen Knoten bei insgesamt 60 verteilten Services

Selbst-Konfiguration

Beide Modelle verleihen dem System die Fähigkeit zur Selbst-Konfiguration. Ziel in diesem Szenario ist es, eine vorgegebene Anzahl an Service-Instanzen zu starten und zu verteilen, so dass ein balancierter und stabiler Zustand erreicht wird.

Es wird wieder ein Netz aus 10 Knoten zu Grunde gelegt. Der Planer auf Knoten 1 erhält als zusätzliches Ziel 20 Service-Instanzen zu starten und zu verteilen. Abbildung 4.5 zeigt die kumulierte Anzahl an Service-Instanzen aller Knoten. Der

Planer startet die gewünschten 20 Service-Instanzen. Das Boolesche Modell benötigt 19 Sekunden während das Numerische Modell nur halb so viel Zeit benötigt und die Aufgabe innerhalb 9 Sekunden löst.

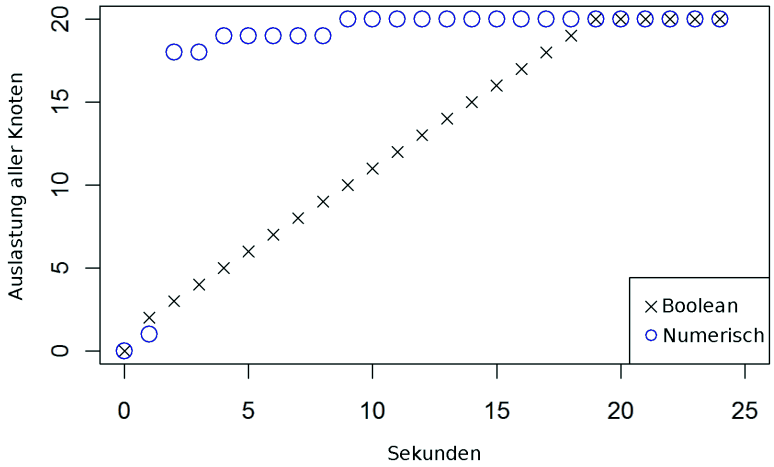


Abbildung 4.5: Selbst-Konfiguration: Kumulierte Anzahl laufender Services

Abbildung 4.6 zeigt die Auslastung von Knoten 1 für die verschiedenen Modelle. Da 20 Services gleichmäßig auf 10 Knoten verteilt werden beträgt die optimale Last pro Knoten 2 Services. Das Boolesche Modell benötigt insgesamt 20 Sekunden um die Services optimal zu verteilen. Das Numerische Modell benötigt für die Verteilung der Services keine zusätzliche Zeit und erreicht nach 9 Sekunden einen stabilen und balancierten Zustand.

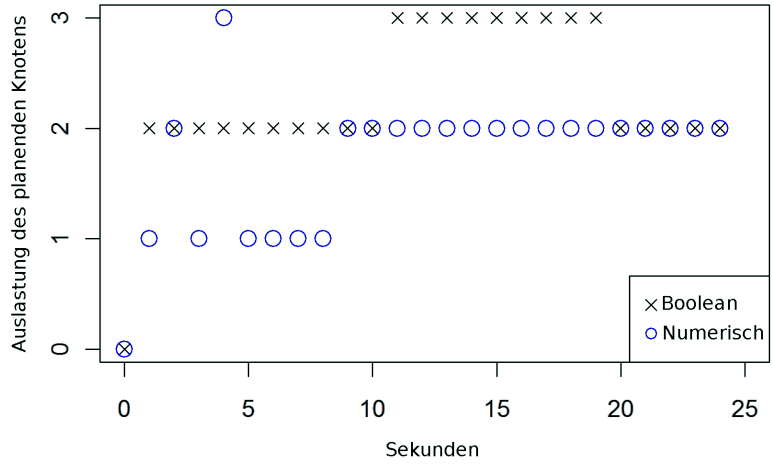


Abbildung 4.6: Selbst-Konfiguration: Auslastung des jeweils planenden Knotens

Selbst-Heilung

Um die Fähigkeiten der Selbst-Heilung zu demonstrieren, wird der Ausfall von Knoten simuliert. Zu Beginn wird ein stabiles und balanciertes Netz von 10 Knoten mit 20 verteilten Services generiert. In diesem System fallen 3 Knoten aus, dadurch laufen 6 Services zu wenig.

In Abbildung 4.7 fällt deshalb die kumulierte Anzahl an Services auf 14 in Sekunde 1. Das Numerische Modell startet alle 6 fehlenden Services innerhalb einer Planungsrunde, wohingegen das Boolesche Modell 6 Sekunden benötigt, um alle fehlenden Services zu starten. Beide Modelle erreichen einen balancierten und stabilen Zustand.

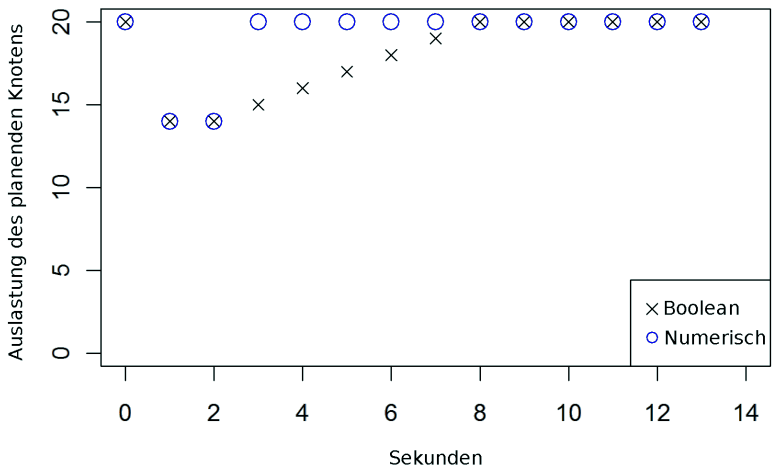


Abbildung 4.7: Ausfall von Knoten bei Sekunde 1

Vergleich der Modelle

Beide Modelle implementieren die gewünschten Selbst-X Eigenschaften Selbst-Optimierung, Selbst-Konfiguration und Selbst-Heilung. In allen Szenarien wurde ein balancierter und stabiler Zustand erreicht. Die Evaluationen zeigen jedoch, dass die vom Booleschem Modell benötigte Zeit abhängig ist von der Anzahl betroffener Service-Instanzen. Je mehr Services gestartet, gestoppt oder verschoben werden sollen, desto mehr Zeit benötigt das Boolesche Modell. Tritt jedoch ein neuer Knoten einem bestehendem Netz bei, treten beim Booleschen Modell weniger Schwankungen in der Auslastung auf als beim Numerischen Modell.

Das Numerische Modell passt sich neuen Situationen sehr schnell an und erreicht den gewünschten Zustand oft innerhalb weniger Planungsrunden. Deshalb wird im weiteren Verlauf nur noch das Numerische Modell betrachtet. In Kapitel 5 findet sich eine ausführlichere Evaluierung dieses Modells.

4.2 Reflex Manager

In jedem AC/OC System existiert eine Komponente, die Entscheidungen trifft um das beobachtete System zu beeinflussen. In OCμ übernimmt diese Aufgabe der Planner Manager. Er verwendet einen automatischen Planer um Pläne zu generieren. Um einen Plan zu erstellen benötigt der Planer eine gewisse Zeit, im Folgenden Planungszeit genannt. Im Allgemeinen gilt: Je komplexer die Eingabedaten des Planers sind und je länger der benötigte Plan ist, umso mehr Planungszeit benötigt ein automatischer Planer. Dies verschlechtert die Reaktionszeit des Systems, also die Zeit zwischen dem Erkennen eines Problems und der Ausführung von Gegenmaßnahmen.

Der Reflex Manager verkürzt die Reaktionszeit des Systems. Er dient als Speicher für bereits erstellte Pläne. Eine Besonderheit des hier vorgestellten Reflex Managers ist es, dass auch Pläne verwendet werden können, welche ursprünglich für eine andere, ähnliche Ausgangssituation erstellt wurden. Dadurch kann die Reaktionszeit des Systems auch verkürzt werden, wenn für den aktuellen Zustand noch gar kein Plan existiert.

Dazu speichert der Reflex Manager vom Planner Manager erstellte Pläne und den jeweiligen Ausgangszustand des Systems ab. Wenn das System den gleichen oder einen ähnlichen Zustand erreicht, kann der Reflex Manager einen gespeicherten Plan auswählen und dem Actuator übermitteln, der den Plan ausführt. Artikel [39] beschreibt das Konzept des Reflex Managers und des Actuators.

4.2.1 Allgemeines Konzept

Der Reflex Manager ergänzt den Planner Manager und ist in der Planning Phase des Organic Managers angesiedelt, wie in Abbildung 4.1 auf Seite 25 dargestellt. Er erhält den Ausgangszustand des Systems von der Fact Base aus der Analyze Phase. Vom Planner Manager erhält der Reflex Manager Pläne, die vom automatischen Planer generiert wurden. Sie werden in der Reflex Base gespeichert. Des Weiteren kann der Reflex Manager Pläne an den Actuator schicken, der sie ausführt.

Der Reflex Manager wandelt den Ausgangszustand von der Fact Base in eine besondere Struktur um, die in Abschnitt 4.2.2 näher erklärt wird. Dadurch ist ein Vergleich zwischen zwei Zuständen möglich. Anschließend untersucht er die gespeicherten Zuständen aus der Reflex Base. Findet er den gleichen oder einen ähnlichen Zustand, so leitet er den zugehörigen Plan an den Actuator weiter.

Der Reflex Manager speichert Pläne in der Reflex Base ab. Tabelle 4.1 zeigt schematisch die Struktur der Reflex Base. Als Schlüssel dient der Ausgangszustand des

Schlüssel	Wert
Zustand A	Plan P_A
Zustand B	Plan P_B
Zustand C	Plan P_C
⋮	⋮

Tabelle 4.1: Reflex Base

Systems, für den der Plan ursprünglich erstellt wurde. Die Reflex Base füllt sich im Laufe der Zeit mit Plänen, die der automatische Planer erstellt hat.

Um die Ressourcen der Knoten nicht zu überlasten wird der Speicherplatz begrenzt. Der Reflex Manager entscheidet welcher Plan gespeichert wird. Ist die maximale Anzahl speicherbarer Pläne erreicht, wird eine Verdrängungsstrategie verwendet um Pläne zu löschen.

4.2.2 Kodierung des Systemzustandes

Um den Ausgangszustand und den Plan des automatischen Planers zu speichern benötigt der Reflex Manager eine geeignete Speicherstruktur. Hierfür verwendet er eine Map mit key/value-Speicherstruktur, bei welcher der Plan als Wert verwendet wird. Der Schlüssel soll flexibel genug sein um neue Informationen, z. B. über einen neuen Service, abspeichern zu können ohne a priori Anpassungen des Reflex Managers vornehmen zu müssen. Als Schlüssel wird der Ausgangszustand, für den der Plan erstellt wurde, verwendet.

Ein Zustand besteht aus einer variablen Anzahl von unterschiedlichen Schlüsselwörtern. Jedem Schlüsselwort ist ein Vektor mit Werten zugeordnet, welche beobachtete Informationen enthalten. Der Name eines Services ist beispielsweise ein solches Schlüsselwort. Sein Vektor besteht aus der Anzahl lokal laufender Instanzen, wie viele Instanzen im kompletten System laufen und ob der Service verschiebbar ist. Ein weiteres Schlüsselwort ist dem lokalen Knoten zugeordnet, in seinem Vektor ist unter anderem die Auslastung des Knotens gespeichert. Weitere bzw. neue Informationen können unter anderen Schlüsselwörtern gespeichert werden. So wird die gewünschte Flexibilität erreicht. Die beschriebene Struktur kann wie folgt dargestellt werden:

- $k \in K$ ein Schlüsselwort aus einer Menge an Schlüsselwörtern K
- $s_k \in \mathbb{R}^{n_k}$ ein Vektor der Länge n_k für das Schlüsselwort k
- $S = \{s_k | k \in K\}$ der Ausgangszustand des Systems, eine Menge von Vektoren mit unterschiedlichen Schlüsselwörtern
- $C = \{c_k | k \in K\}$ der aktuelle Zustand des Systems

4.2.3 Planauswahl

Der Reflex Manager hat mehrere Zustände $A, B, \ldots$ gespeichert und verwendet sie als Schlüssel, um einzelne Pläne abzuspeichern, wie in Abbildung 4.8 dargestellt. Um für den aktuellen Zustand S einen passenden Plan zu finden, muss der Reflex Manager in der Lage sein, Zustände miteinander zu vergleichen. Da die gespeicherten Pläne nicht nur bei exakt gleichem Zustand sondern auch bei hinreichend ähnlichem Zustand wieder verwendet werden sollen, wird für den Vergleich eine Abstandsmetrik verwendet.

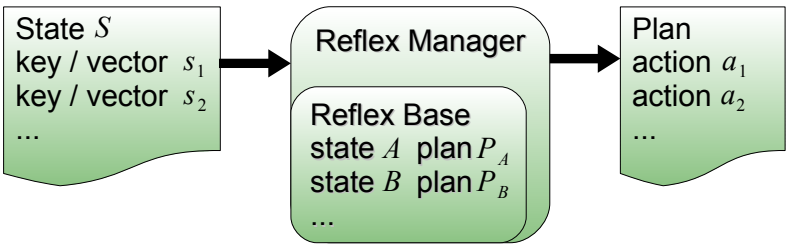


Abbildung 4.8: Reflex Manager

Im Folgenden werden zwei Metriken vorgestellt. Die erste Metrik basiert auf dem euklidischen Abstand und wurde leicht angepasst, um mit der vorliegenden Speicherstruktur umgehen zu können:

Definition 2 (Metrik 1)

$$|S - C| = r(S, C) + \sum_{k \in S \cap C} \sqrt{\sum_{i}^{n_k} (s_{k,i} - c_{k,i})^2}$$

Der Term unter der Wurzel entspricht dem euklidischem Vorbild. In der Summe vor der Wurzel werden jedoch nur jene Indizes ausgewertet, die in beiden Mengen S und C enthalten sind, d h. es werden nur Schlüsselwörter verwendet, die in beiden Zuständen vorkommen.

Definition 3 (Metrik 2)

$$|S - C|_\lambda = r(S, C) + \sum_{k \in S \cap C} \sqrt{\sum_{i}^{n_k} \lambda_{k,i} (s_{k,i} - c_{k,i})^2}$$

Die zweite Metrik verwendet einen Gewichtungsfaktor $\lambda_{k,i} \in \mathbb{R}$, der einen Vektor $\lambda_k \in \mathbb{R}^{n_k}$ mit $k \in K$ jedem Schlüsselwort zuordnet. Jede Wertedifferenz unter der Wurzel wird mit einem Faktor gewichtet. Dies kann dazu verwendet werden, verschiedene Wertebereiche der beobachteten Daten zu berücksichtigen. So kann

beispielsweise die Differenz von Booleschen Werten gegenüber der Differenz von Prozentwerten oder natürlichen Zahlen aufgewertet werden, um den Effekt der Booleschen Werte auf das Ergebnis zu verstärken.

Der Wert $r(S, C) \geq 0$ erhöht den Abstand zwischen S und C. Falls beide Mengen genau die gleichen Schlüsselworte enthalten, ist $r(S, C) = 0$. Anderenfalls gibt es abhängig von der verwendeten Strategie verschiedene Möglichkeiten $r(S, C)$ zu wählen:

- $r(S, C) = 0$ falls ein zusätzliches oder fehlendes Schlüsselwort in S oder C keinen Einfluss auf die Distanz hat
- $r(S, C) = \infty$ falls die Zustände bei einem unterschiedlichen Schlüsselwort komplett verschieden sind
- $r(S, C) = \sum_{k \in (S \setminus C \cup C \setminus S)} \alpha$ für jedes zusätzliche bzw. fehlendes Schlüsselwort wird die Distanz um einen fixen Wert $\alpha > 0$ erhöht

4.2.4 Verdrängungsstrategien

Eine Verdrängungsstrategie ist notwendig, wenn ein neuer Plan gespeichert werden soll und alle vorhandenen Speicherplätze bereits belegt sind. Die Verdrängungsstrategie hat Zugriff auf den neuen Plan und die Reflex Base. Sie entscheidet, ob der neue Plan gespeichert wird und welcher alte Plan dafür gelöscht wird.

- FiFo: Das älteste Element wird gelöscht
- LiFo: Das jüngste Element wird gelöscht
- LFU: Das am wenigsten verwendete Element wird gelöscht
- LRU: Das am längsten nicht verwendete Element wird gelöscht
- Random: Ein zufälliges Element wird gelöscht
- Shortest: Der kürzeste Plan wird gelöscht
- Longest: Der längste Plan wird gelöscht
- Fastest: Der Plan mit kürzester Erstelldauer wird gelöscht
- Similar: Der ähnlichste Zustand wird gelöscht

Bei *First in First out (FiFo)* wird das älteste Element gelöscht. Der Systemzustand enthält den Zeitpunkt, zu dem er gemessen wurde. So wird ermittelt, welcher Zustand der älteste ist und gelöscht werden muss.

Eine weitere bekannte Strategie ist *Last in First out (LiFo)*. Hierbei wird das jeweils neueste Element gelöscht. Dies bedeutet, dass immer der als letztes hinzugefügte Zustand gelöscht wird und die Reflex Base sonst nur alte Zustände enthält. Die Reflex Base enthält somit nur einen neuen Plan, der in jedem Planungsschritt überschrieben wird.

Bei *Least Frequently Used (LFU)* wird das Element gelöscht, welches am wenigsten verwendet wird. Zu jedem Zustand wird dokumentiert, wie oft er ausgewählt wurde. Neue Pläne werden naturgemäß weniger oft verwendet als alte. Dadurch besteht die Gefahr, dass die Strategie sich FiFo annähert, was nicht erwünscht ist. Um dem entgegen zu wirken, kann eine Bewährungsphase für neue Pläne eingeführt werden. Neue Pläne werden erst nach dieser Zeit in die Auswahl der löschbaren Elemente aufgenommen. Eine andere Möglichkeit besteht darin, die Anzahl der Verwendung abhängig von der Zeit zu gestalten.

Least Recently Used (LRU) löscht das am längsten nicht verwendete Element. Zu jedem Zustand wird der Zeitpunkt der letzten Verwendung vermerkt. Das am längsten nicht verwendete Element wird gelöscht. Diese Strategie zahlt sich bei oft wiederkehrenden Zuständen aus.

Ein zufälligen Zustand löscht die Verdrängungsstrategie *Random*. Der zu speichernde Plan wird dabei nicht beachtet.

Shortest und *Longest* sind gegensätzliche Strategien. Beide verwenden die Anzahl Aktionen um die Länge eines Planes zu bestimmen. Während *Shortest* den kürzesten Plan löscht, wählt *Longest* den längsten Plan aus. Bei kurzen Plänen ist die Chance größer, auf den Plan des Planers zu wechseln.

Fastest löscht den Plan, dessen Erstelldauer am geringsten war. So bleiben komplizierte und lange Pläne erhalten.

Die Verdrängungsstrategie *Similar* vergleicht alle Zustände miteinander. Der Zustand mit dem geringsten Abstand zu einem anderen Zustand wird gelöscht. Ziel ist es, eine möglichst hohe Abdeckung aller möglichen Zustände zu erhalten und in vielen Situationen einen Treffer zu erhalten. Dabei wird davon ausgegangen, dass alle Zustände wiederkehren können.

Eine Evaluierung und Vergleich der Verdrängungsstrategien findet sich in Abschnitt 7.3.

4.3 Actuator

In diesem Abschnitt wird die Funktionsweise des Actuators näher beleuchtet. Abschnitt 4.3.1 befasst sich mit konkurrierenden Plänen des Planner Managers und des Reflex Managers.

Der Actuator ist für die Ausführung von Plänen zuständig. Ein Plan besteht aus einer geordneten Folge von Aktionen. Der Actuator identifiziert die Aktion durch ihren Namen und führt die Aktionen hintereinander aus. Dabei können manche Aktionen entkoppelt sein, wenn z. B. ein Service auf einem anderen Knoten gestartet wird. Somit kann eine Aktion noch ausgeführt werden, während der Actuator bereits die nächste Aktion anstößt.

Zudem hat der Actuator die Möglichkeit, auf aktuelle Informationen aus dem Information Pool zurück zu greifen. Wenn beispielsweise ein Service verschoben werden soll, nutzt der Actuator den Information Pool um heraus zu finden, welcher Knoten am besten geeignet ist um den Service aufzunehmen. Wenn eine gleichmäßige Auslastung erzielt werden soll, wählt der Actuator den Knoten mit der aktuell geringsten Auslastung.

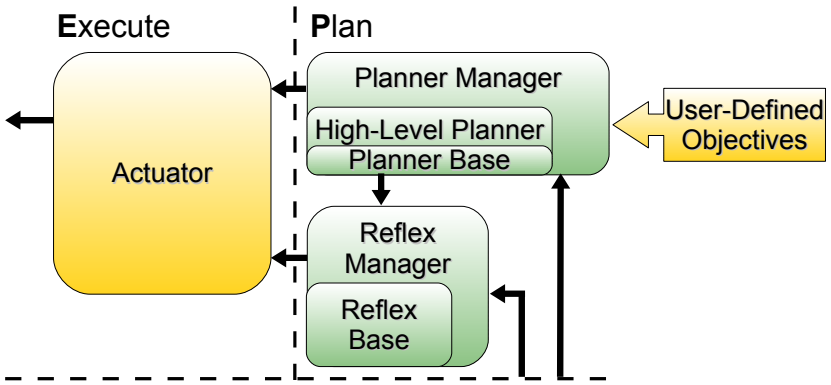


Abbildung 4.9: Execute und Plan Phase

Wie in Abbildung 4.9 erkennbar, ist er in der letzten Phase des MAPE Zyklus des Organic Managers angesiedelt. Er erhält Pläne vom Planner Manager und vom Reflex Manager. Im Allgemeinen können folgende drei Fälle unterschieden werden:

1. Der Planner Manager leitet seinen Plan zuerst an den Actuator weiter.

2. Der Reflex Manager war schneller und der Plan ist bereits ausgeführt.

3. Der Reflex Manager war schneller und der Plan ist teilweise ausgeführt.

Im ersten Fall kann davon ausgegangen werden, dass der Reflex Manager noch keinen Plan für die aktuelle Situation gespeichert hat. In diesem Fall kann der Actuator den Plan direkt ausführen.

Im zweiten Fall ist der Plan des Reflex Managers bereits ausgeführt. Da in einem verteilten, Organic System ein Rollback schwierig und teilweise nicht möglich ist, wird ein solches Verhalten vom Actuator nicht unterstützt. Stattdessen wird der im nächsten Abschnitt beschriebene Vergleich ausgeführt. Schlägt er fehl, erhält der Planer in der nächsten Planungsrunde die Chance noch nötige Aktionen zu bestimmen, falls notwendig.

4.3.1 Konkurrierende Pläne

Der Fokus in diesem Abschnitt liegt auf dem dritten Fall. Nachdem der Reflex Manager und der Planner Manager angestoßen wurden leitet der Reflex Manager zuerst seinen Plan R an den Actuator weiter. Während der Ausführung der n-ten Aktion erhält der Actuator einen Plan P vom Planner Manager. Der Actuator hält die Ausführung des Planes an und vergleicht die beiden Pläne. Wurde der Plan des Reflex Managers bereits komplett ausgeführt, so wird auch ein Vergleich vorgenommen. In diesem Fall entspricht n der Anzahl an Aktionen des Reflex Manager Plans.

Jeder Plan besteht aus einzelnen, geordneten Aktionen. Die Aktionen $r_1 \dots r_n$ wurden bereits vom Actuator ausgeführt. Abbildung 4.10 zeigt ein einfaches Beispiel mit drei Aktionen $\{a, b, c\}$. Im besten Fall stimmen die n bereits ausgeführten Aktionen exakt mit den ersten n Aktionen des Plans P vom Planner Manager überein: $\forall i$ mit $1 \leq i \leq n : r_i = p_i$. In diesem Fall kann der Actuator zu Plan P wechseln und die Aktionen $p_{n+1}, \dots$ ausführen.

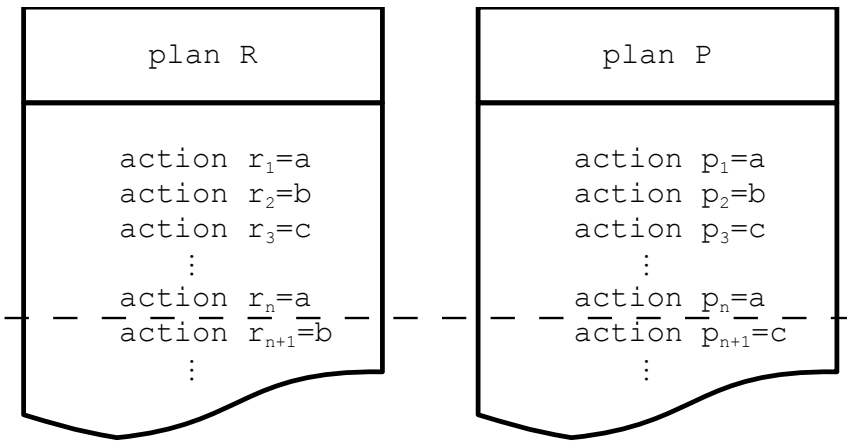


Abbildung 4.10: Vergleich von zwei Plänen

Falls keine bedingten Effekte verwendet werden kann unter Umständen auch bei vertauschten Aktionen der Plan gewechselt werden. Bei bedingten Effekten wird ein Wert nur geändert, wenn ein Prädikat zutrifft, wie beispielsweise:

```
when(sonnig) (Solarstrom = true)
```

Bedingte Effekt können vermieden werden indem die betreffende Aktion in mehrere Aktionen aufgeteilt wird, so dass eine Aktion die Bedingung bereits als Vorbedingung enthält und der betreffende Wert als Effekt verändert wird.

Eine Vertauschung kann ignoriert werden, wenn der Folgezustand auch bei einer Vertauschung derselbe ist. Dies ist beispielsweise der Fall, wenn die Menge an Prädikaten und Funktionen, auf die sich die Aktionen auswirken, disjunkt ist. Wird eine gemeinsame Funktion nur relativ verändert, also beispielsweise um zwei erhöht oder verringert, ändert eine Vertauschung auch nichts. Die in dieser Arbeit verwendete Modelle enthalten nur Aktionen, bei denen eine Vertauschung keine Änderung des Folgezustands nach sich ziehen.

Falls keine bedingten Effekte verwendet werden und eine Vertauschung keine unterschiedlichen Folgezustände kreiert, kann folgende Situation gelöst werden. Die ersten n ausgeführten Aktionen stimmen mit den ersten n Aktionen des Plans P überein, haben jedoch eine andere Reihenfolge. Wie in Abbildung 4.11 angedeutet, stimmt jede Aktion $\{r_1, \ldots, r_n\}$ mit einer Aktion aus $\{p_1, \ldots, p_n\}$ genau überein. In diesem Fall kann der Actuator zu Plan P wechseln und diesen Plan zu Ende führen.

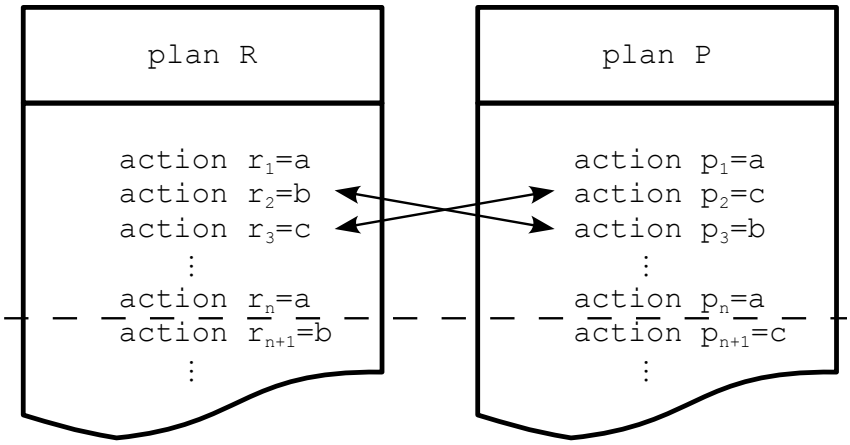


Abbildung 4.11: Vertauschte Reihenfolge der Aktionen

5 Evaluierung

Dieses Kapitel beschreibt die Evaluierung von OCμ mit komplett implementierten Organic Manager. Dem Planer liegt das in Abschnitt 4.1.3 beschriebene Numerische Modell zu Grunde. Gezeigt werden soll, dass alle Selbst-x Eigenschaften sowie die Ziele der Anwender durch den Organic Manager erfüllt werden.

5.1 Evaluierungskriterien

Die Ergebnisse der Evaluierungen werden in den folgenden Abschnitten auf mehrere Kriterien hin untersucht. Das System soll verschiedene Selbst-X Eigenschaften erfüllen. Im Folgenden wird erklärt, welche Ausprägungen diese Eigenschaften in den vorliegenden Evaluierungen annehmen.

Selbst-Optimierung: Alle Knoten sollen über eine gleichmäßige Auslastung verfügen. Die Auslastung wird durch die Anzahl an Services auf einem Knoten gemessen. Zu beachten ist hierbei, dass oftmals nicht alle Knoten genau die gleiche Auslastung haben können. Die Anzahl an Services ist ganzzahlig, daher kann es sein, dass auf einem Knoten ein Service mehr aktiv ist als auf einem anderen.

Selbst-Konfiguration: OCμ soll alle erforderlichen Services starten oder stoppen, falls nötig. Hierbei handelt es sich insbesondere um Services, welche vom Anwender vorgegeben werden. Das System muss die vorgegebene Anzahl an erforderlichen Services einhalten.

Selbst-Heilung: Das System soll selbstständig erkennen, dass ein Service ausgefallen ist. Ausgefallene Services werden automatisch neu gestartet.

Neben diesen Selbst-X Eigenschaften sind noch andere Eigenschaften des Systems von Interesse. Insbesondere soll gezeigt werden, dass die Zweiteilung der Planungsphase in Planner Manager und Reflex Manager von Vorteil ist.

5.2 Cloud-Computing-Szenario

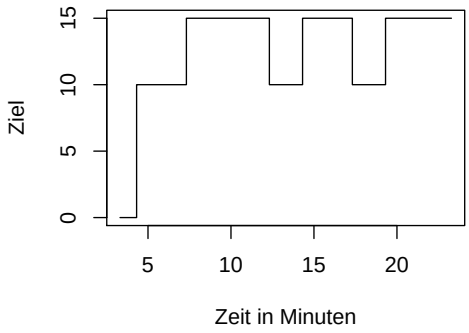
In diesem Szenario stellt OCμ ein Cloud-Computing-System dar. OCμ dient als Plattform, auf der Anwender ihre Programme entwickeln und ausführen können.

Das folgende Cloud-Computing-Szenario (CCS1) basiert auf zehn Knoten, auf denen jeweils eine OCμ Instanz läuft. Die Knoten überwachen sich gegenseitig,

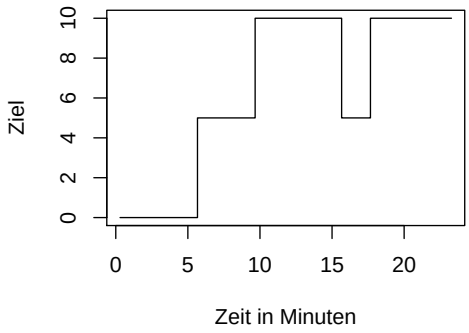
mittels des in [36] beschriebenen Mechanismus. Der maximale Abstand zwischen zwei Nachrichten beträgt hierbei fünf Sekunden. Der Planner Manager wird alle 30 Sekunden angestoßen. Der Reflex Manager verwendet FiFo als Verdrängungsstrategie und speichert bis zu 50 Pläne. Das Szenario erstreckt sich über einen Zeitraum von 25 Minuten.

Auf allen zehn Knoten ist die Selbst-Optimierung aktiviert, die Arbeitslast soll möglichst gleichmäßig auf die Knoten verteilt werden. Auf drei Knoten werden dynamisch weitere Ziele hinzugefügt.

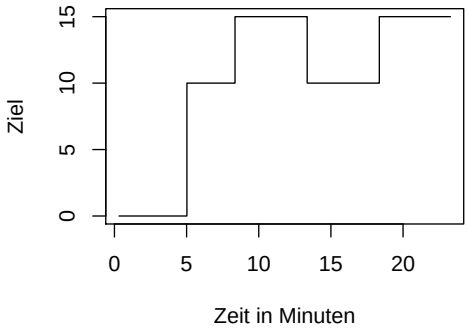
Zu Beginn startet jeder dieser drei Knoten 20 Services. Diese 60 Services stellen die Grundlast des Systems dar. Sie symbolisieren notwendige Systemservices oder Services von weiteren Cloud Anwendern. Jeder der drei Knoten fügt seinem jeweiligen Planer individuelle Ziele hinzu und ändert diese während der Laufzeit. Dies spiegelt das Verhalten von drei verschiedenen Anwendern wieder.



(5.1.a) Ziel Service A1 Anwender 1



(5.1.b) Ziel Service A2 Anwender 2



(5.1.c) Ziel Service A3 Anwender 3

Abbildung 5.1: CCS1: Ziele verschiedener Anwender

Das Ziel des ersten Anwenders ist in Abbildung 5.1.a dargestellt. Zuerst möchte er zehn Services A1 verteilt im Netz laufen lassen. Später erhöht er diese Anzahl auf 15. Nach einigen Minuten setzt er die gewünschte Anzahl wieder auf 10. Er verändert sein Ziel noch weitere dreimal und ist damit der Anwender mit den meisten Ziel-Änderungen.

Fünf Services des Typs A2 setzt der zweite Anwender auf einem anderen Knoten als Ziel fest. Später erhöht er dieses Ziel auf zehn Services, wie in Abbildung 5.1.b dargestellt. Im weiteren Zeitverlauf verringert er kurz die gewünschte Anzahl an Service auf fünf um sie anschließend wieder auf zehn zu setzen.

Abbildung 5.1.c zeigt das Ziel des dritten Anwenders. Er will zehn Services des Typs A3 verteilt laufen lassen. Später erhöht er die gewünschte Anzahl auf 15. Nach einigen Minuten ändert er sein Ziel wieder auf zehn ab. Zuletzt soll die Anzahl an Services wieder 15 betragen.

Um die Selbst-Heilung zu demonstrieren ist auf jedem Knoten ein eigenes Programm aktiv, das Services beendet. Alle 30 Sekunden beendet es mit Wahrscheinlichkeit $0,5$ einen Service der Anwender.

5.3 Evaluierungsergebnisse

Die folgenden Evaluierungsergebnisse sollen zeigen, dass OCμ mit Hilfe des Organic Managers in einem Cloud-Computing-Szenario fähig ist Selbst-Heilung, Selbst-Optimierung und Selbst-Konfiguration umzusetzen und zusätzlich Ziele des Anwenders zu berücksichtigen. Um dies zu verdeutlichen werden verschiedene Evaluationsergebnisse graphisch dargestellt. Hierbei werden die zehn Knoten farbig markiert. Die Farbe eines spezifischen Knotens ist in allen Abbildungen innerhalb dieser Evaluierung gleich. Die folgenden Kapitel behandeln eine einzelne Evaluierung des in Kapitel 5.2 vorgestellten Szenarios.

5.3.1 Selbst-Optimierung

Die Selbst-Optimierung hat in diesem Szenario die Aufgabe, die anfallende Last gleichmäßig zu verteilen. Dies bedeutet, dass die Knoten eine möglichst gleiche Anzahl an Services ausführen sollen. Abbildung 5.2 zeigt die Anzahl an laufenden Services. Jede Linie stellt dabei die Auslastung eines einzelnen Knotens dar. Zu Beginn der Evaluierung startet jeder Knoten fünf Services, die Basisfunktionalitäten enthalten. Auf drei Knoten werden jeweils 20 Services B1 gestartet. Die Organic Manager dieser drei Knoten verteilen diese Services anschließend gleichmäßig auf alle anderen Knoten. Durch das Stoppen und Starten von Services ergeben sich

immer wieder Schwankungen, jedoch kann man deutlich erkennen, dass sich die Auslastung der Knoten einander annähert bzw. gleich ist.

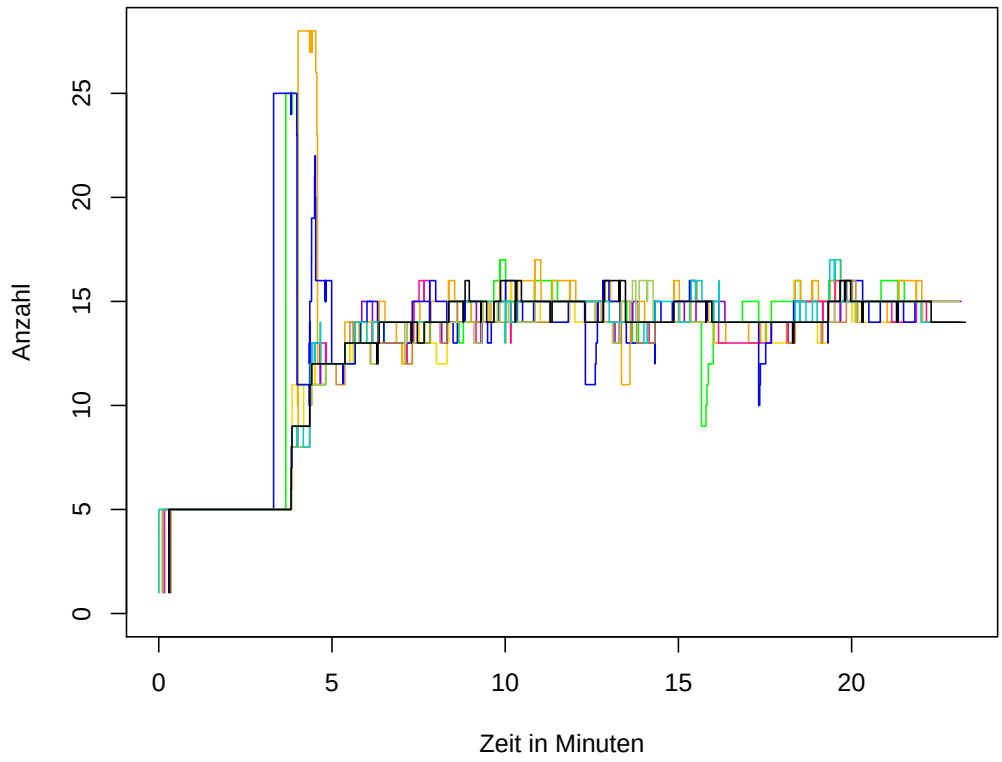


Abbildung 5.2: CCS1: Summe Services je Knoten

Das Stoppen von Services hat eine größere Auswirkung als das Starten von Services. Beim Starten von Services kann der Planer frei entscheiden, ob der Service lokal oder auf einem anderen Knoten gestartet werden soll. Beim Stoppen jedoch hat der Planer weniger Entscheidungsfreiheit. Er kann nur Services auf Knoten stoppen, die tatsächlich laufen. In Abbildung 5.1.c kann man erkennen, dass sich der Nutzer kurz nach Minute 15 dazu entscheidet, die Anzahl an Services um fünf zu verringern. Die Services werden hauptsächlich auf einem Knoten gestoppt. Deshalb kann man in Abbildung 5.2 kurz nach Minute 15 einen Einbruch in der Auslastung des hellgrünen Knoten erkennen. Die anderen Knoten bemerken dies jedoch und verschieben Services zu diesem Knoten. Ebenso erklären sich die anderen drei kleinen Einbrüche nach unten zwischen Minute 10 und 20.

5.3.2 Ziele der Anwender und Selbst-Heilung

In den folgenden Abbildungen wird die Anzahl an Services im Zeitverlauf dargestellt. Die Zielanzahl, die der Anwender vorgibt, ist mit einer schwarzen Linie gekennzeichnet. Die unterschiedlichen Farben zeigen an, auf welchem Knoten der Service läuft. Die Kreuze am oberen Rand der Abbildung signalisieren, dass ein Service ausgefallen ist. In manchen Abbildungen fallen schmale Strich nach oben oder unten auf. Sie entstehen vor allem, wenn ein Service von einem Knoten auf einen anderen verschoben wird. Dabei wird der Service erst auf einem Knoten gestoppt bevor er verschoben und auf einem anderen wieder gestartet wird. Im Regelfall zeigt sich dies in den Abbildungen durch einen schmalen Strich nach unten. Es kann jedoch vorkommen, dass aufgrund der beschränkten Genauigkeit der Zeitmessung in Java oder nicht synchroner Uhren ein Strich nach oben entsteht.

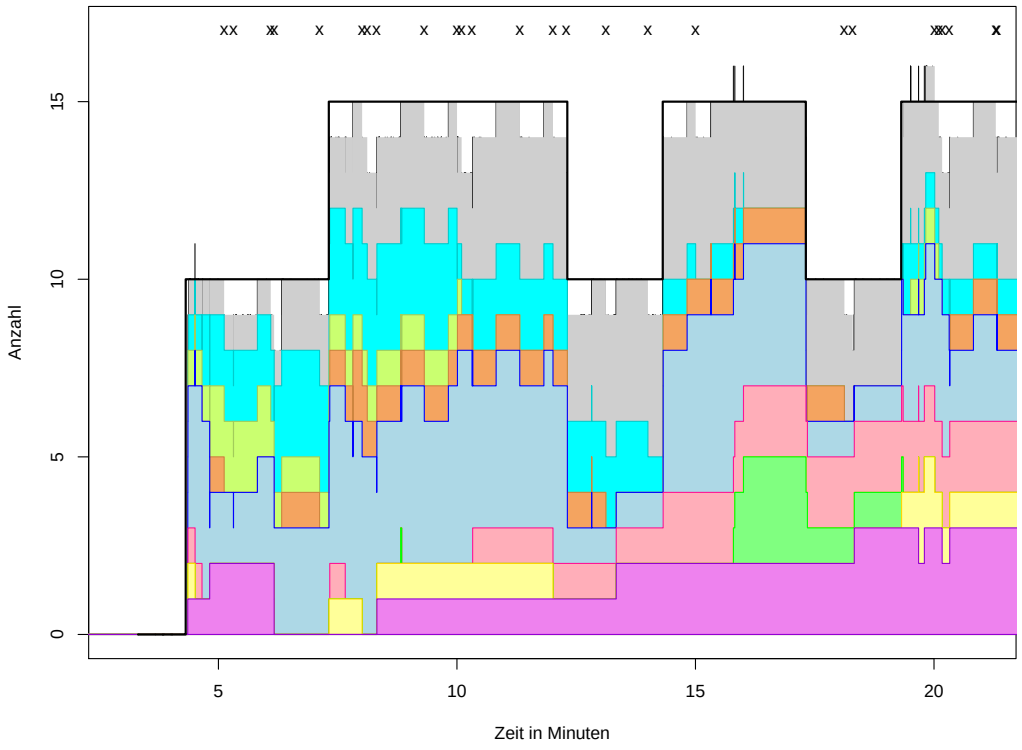


Abbildung 5.3: CCS1: Summe aller Services A1

Abbildung 5.3 zeigt die Anzahl laufender Services des Typs A1. Dem Wunsch des Anwenders entsprechend laufen abwechselnd 10 bzw. 15 Services A1. Der Organic Manager verteilt die Services auf verschiedene Knoten, welche durch die farbigen Streifen dargestellt sind. Da noch andere Services im System verteilt laufen,

erhalten weniger ausgelastete Knoten mehr Services und andere, mehr ausgelastete Knoten keinen Service.

Nachdem ein Service auf irgendeinem Knoten beendet wurde benötigt das System Zeit, um diese Information an den Knoten des betreffenden Anwenders weiterzuleiten. Dort erhält der Organic Manager diese Information und kann einen neuen Service starten. Der automatische Planer wird alle 30 Sekunden angestoßen. Deshalb kann man nach Beendigung eines Services einen kurzen Einbruch der Anzahl an Services sehen. Trotz der häufigen Störungen kann der Planer das System heilen und die Wunschanzahl an Services A1 erreichen.

Abbildung 5.4 zeigt die Anzahl Services A2, die von den Zielen des zweiten Anwenders abhängig ist. Wie man an den Streifen erkennen kann, werden die Services auf weniger Knoten verteilt als beim ersten Anwender. Zum einen ist ihre Anzahl geringer, zum anderen hat der Organic Manager entschieden, einige Services auf dem hellgrünen Knoten zu starten, um ein ausgeglichenes System zu erreichen. Durch die Verteilung auf weniger Knoten, werden in dieser Evaluation auch nicht so viele Services A2 beendet wie A1. Der Organic Manager muss das System seltener heilen und so kann die Wunschanzahl des Anwenders öfters eingehalten werden.

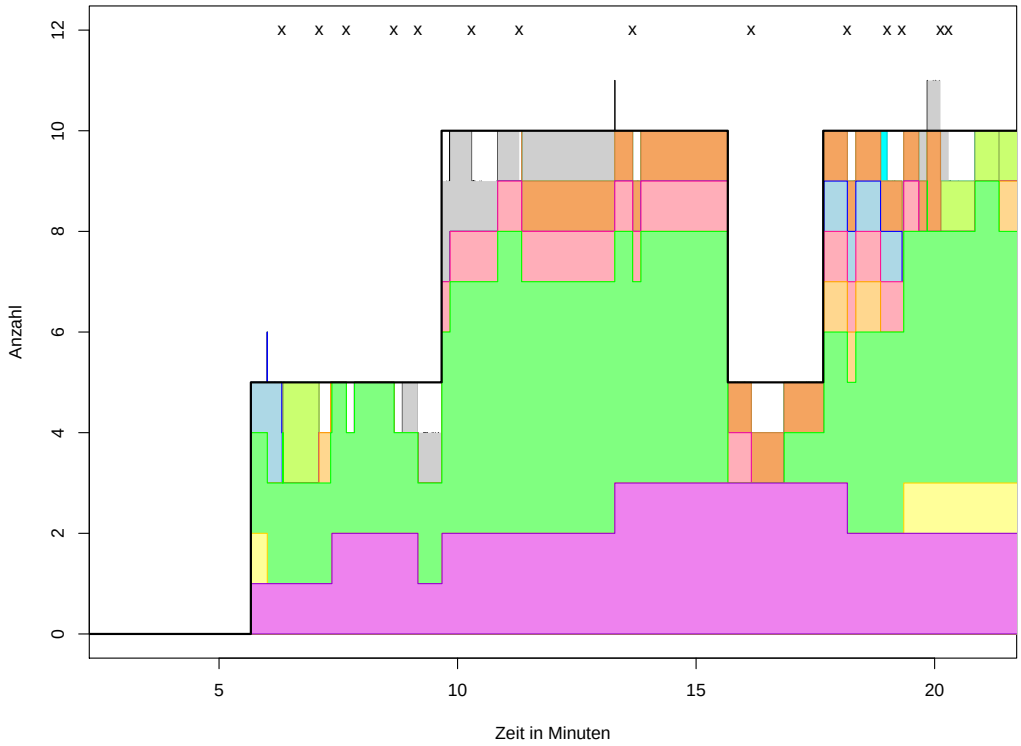


Abbildung 5.4: CCS1: Summe aller Services A2

Der dritte Anwender möchte 10 bzw. 15 Services A3 laufen lassen, wie in Abbildung 5.5 erkennbar. Eine Besonderheit ist die Lücke zu Beginn der Evaluation. Hier wurde das Ziel dem Oranic Manager hinzugefügt, da der Planer jedoch nur alle 30 Sekunden plant, dauert es kurz bis das System reagiert. Die Services wurden wieder auf viele Knoten verteilt und deshalb besonders oft beendet. Da zufälligerweise auf mehreren Knoten in sehr kurzem Abstand je ein Service A3 gestoppt wurde, ist der Einbruch der Anzahl an Services deutlich zu erkennen.

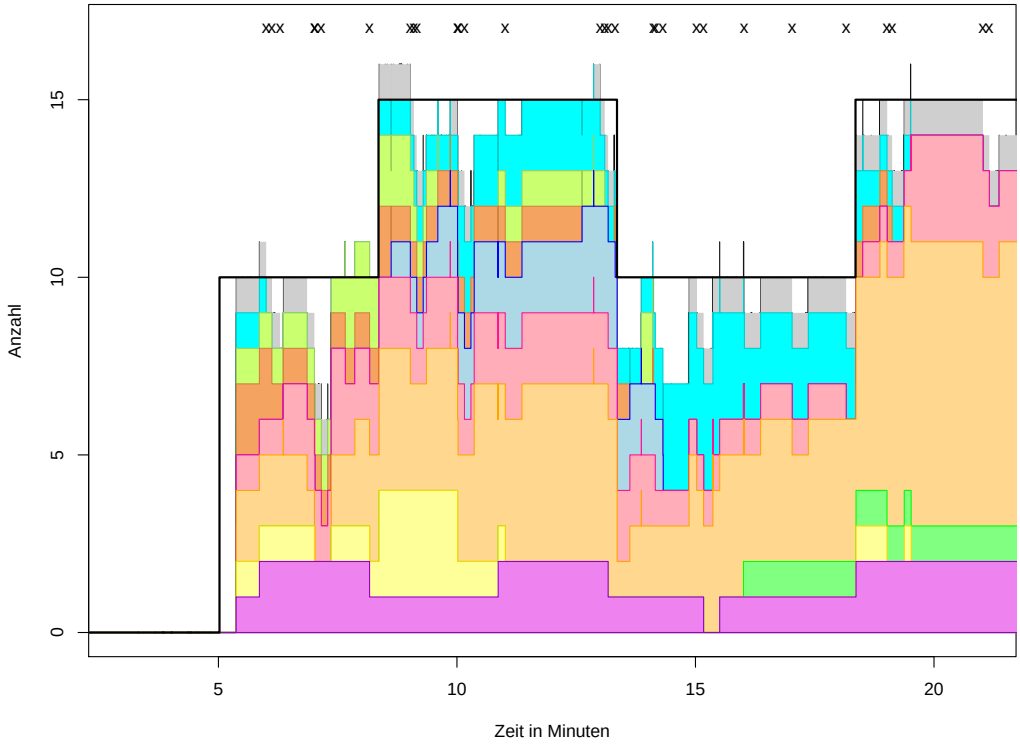


Abbildung 5.5: CCS1: Summe aller Services A3

Alle drei Ziele der Anwender wurden durch die jeweiligen Organic Manager umgesetzt. Der Planer hat die Selbst-Heilung umgesetzt und nach jedem Ausfall eines Services das System in einen gültigen Zustand überführt.

5.3.3 Planner Manager

In der vorliegenden Evaluierung verfolgen drei Knoten Ziele der Anwender und haben somit einen erhöhten Planungsbedarf. Dies spiegelt sich auch in der Länge der Pläne wieder, welche in Abbildung 5.6 als Box-Whisker-Plot dargestellt sind. Die schwarzen Balken entsprechen dem Median der Daten. Die Box um den Median beginnt bei der 25% Quartile und endet bei der 75% Quartile. D. h. sie umfasst 50% der Daten. Die Länge der sich anschließenden Antennen wird durch den eineinhalbfachen Abstand zwischen diesen Quartilen festgelegt. Die Antennen werden jedoch nur bis zu dem Wert gezeichnet, welcher tatsächlich in den Daten vorhanden ist. Werte außerhalb der Antennen werden als Punkte dargestellt. Die sieben Knoten ohne aktiven Anwender müssen nur kleine Anpassungen vornehmen und erzeugen deshalb auch nur kurze Pläne. Die anderen drei Knoten jedoch müssen zu Beginn viele Verschiebungen vornehmen und später auf die wechselnden Ziele ihrer Anwender reagieren. Deshalb haben Knoten drei, vier und sechs Pläne mit einer höheren Anzahl Aktionen.

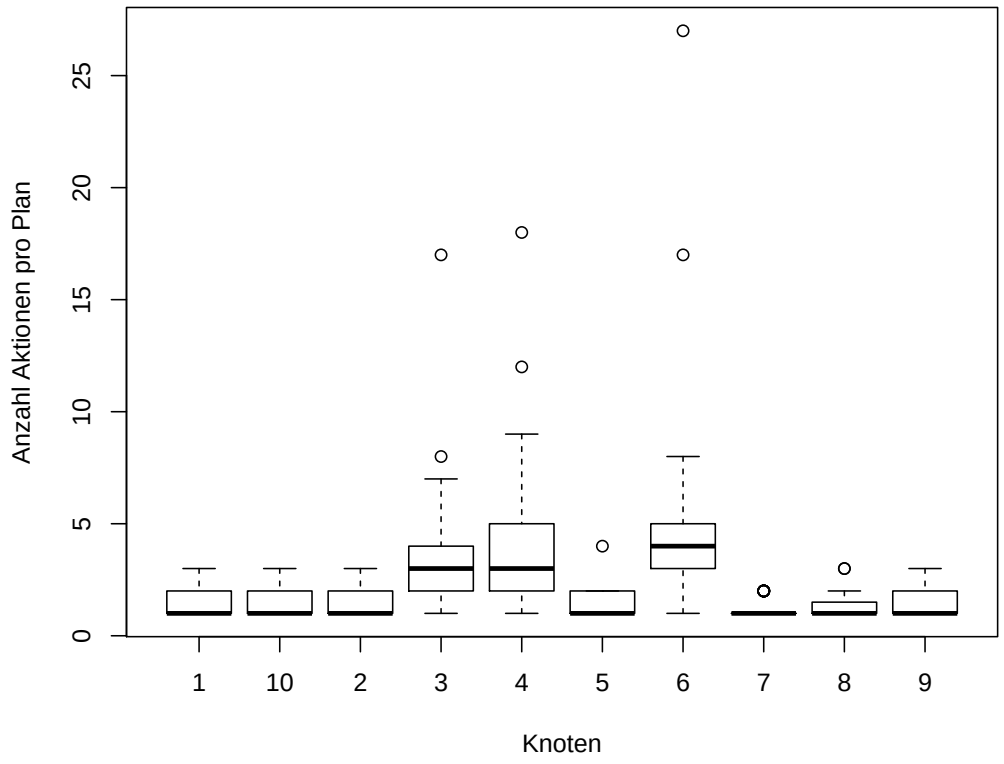


Abbildung 5.6: CCS1: Planlänge

5.3.4 Reflex Manager

In der Reflex Base werden die Pläne des Planers abgespeichert. Werden die Ziele eines Knotens geändert, werden alle Pläne der Reflex Base gelöscht, da sie nicht mehr Ziel führend sind. Abbildung 5.7 zeigt die Anzahl der gespeicherten Pläne der zehn verschiedenen Knoten. Die drei unteren Linien gehören zu den drei planenden Knoten auf welchen die Anwender die Ziele verändern. Durch das häufige Hinzufügen von Zielen wird die Reflex Base oft gelöscht und enthält nur wenige Pläne.

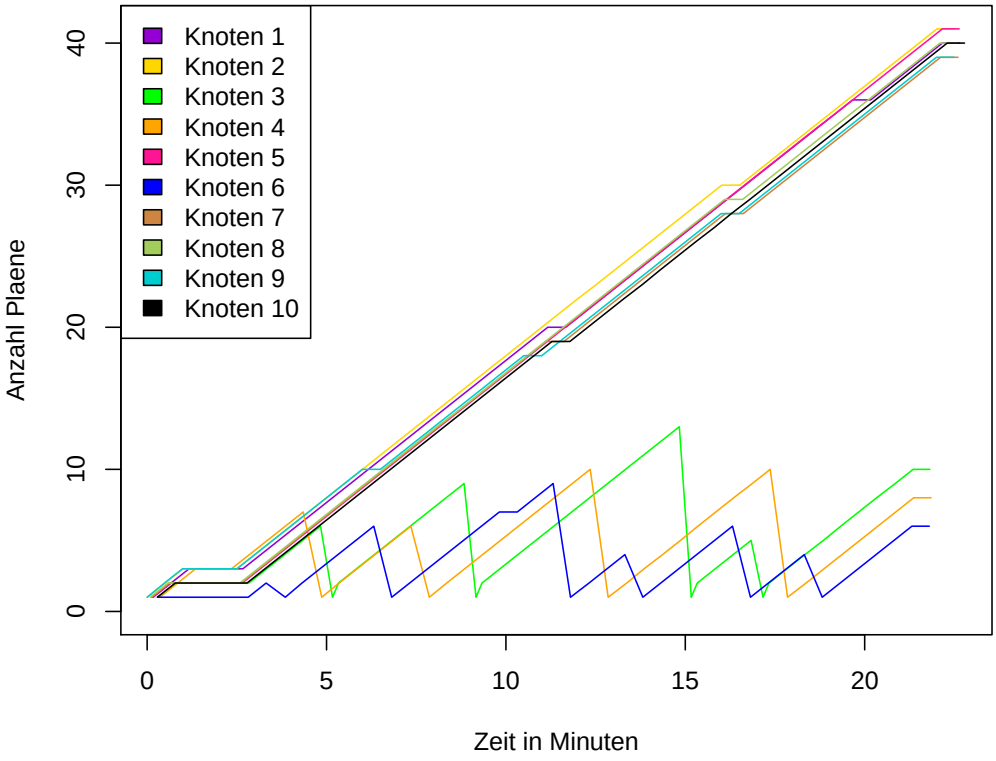


Abbildung 5.7: CCS1: Größe Reflex Base

Abbildung 5.8 zeigt die Trefferraten des Reflex Managers in Prozent. Das entspricht der Anzahl Pläne, die der Reflex Manager ausgewählt hat im Verhältnis zu allen Plänen. Auffällig ist dabei, dass die drei Knoten mit Zielen der Anwender nicht besonders hervorstechen, obwohl ihre Reflex Base immer wieder gelöscht wird. Bei der vorliegenden Evaluierung muss der Actuator keine konkurrierende Pläne auflösen, da der Planer immer denselben Plan wie der Reflex Manager empfiehlt.

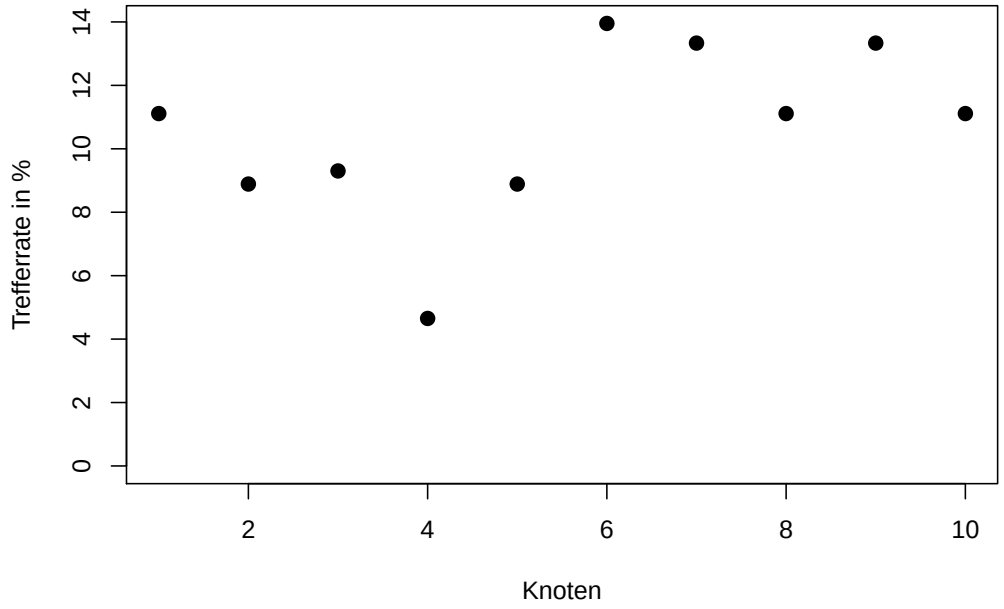


Abbildung 5.8: CCS1: Trefferrate der Reflex Base

5.3.5 Vergleich Planer vs. Reflex Manager

Die Planungsphase des Organic Managers liefert für die aktuell beobachtete Situation einen passenden Plan. Innerhalb dieser Phase gibt es zwei Komponenten, welche die Fähigkeit besitzen einen Plan auszuwählen. Der Planner Manager erstellt einen Plan unter Verwendung eines automatischen Planers. Der Reflex Manager greift auf eine Menge gespeicherter Pläne zurück und vergleicht nur den beobachteten Zustand des Systems mit gespeicherten Zuständen. Abbildung 5.9 zeigt die Zeit, welche der Planer bzw. der Reflex Manager benötigt, um einen Plan zu erstellen bzw. auszuwählen. Dabei wurden die Daten aller Knoten verwendet und als Boxplots dargestellt. Hier kann man nur erkennen, dass der Planer einige Ausreißer hat, die deutlich höher sind als die Zeiten des Reflex Managers.

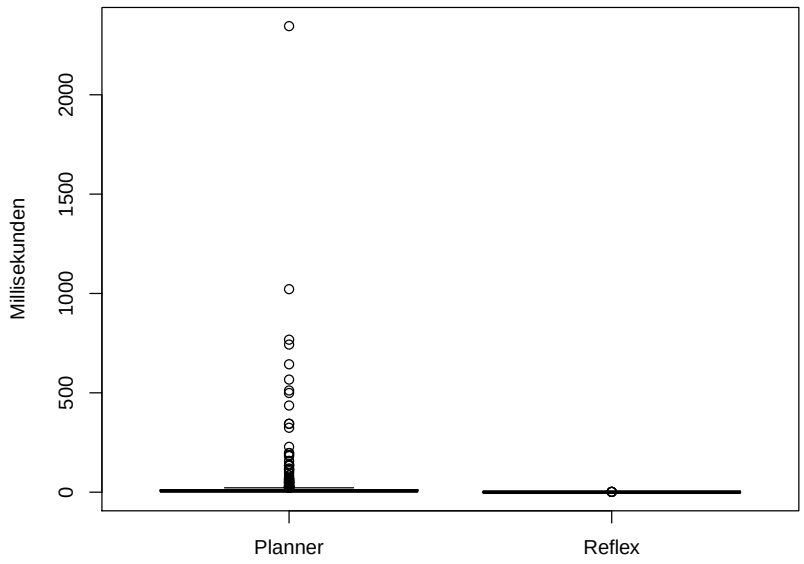


Abbildung 5.9: CCS1: Vergleich Planner Manager und Reflex Manager

Abbildung 5.10 betrachtet nur Zeiten kleiner 25 Millisekunden. Die Werte des Reflex Managers zeigen auch Ausreißer auf, diese sind jedoch immer geringer, als die kürzeste Zeit die der Planer benötigt um einen Plan zu erstellen.

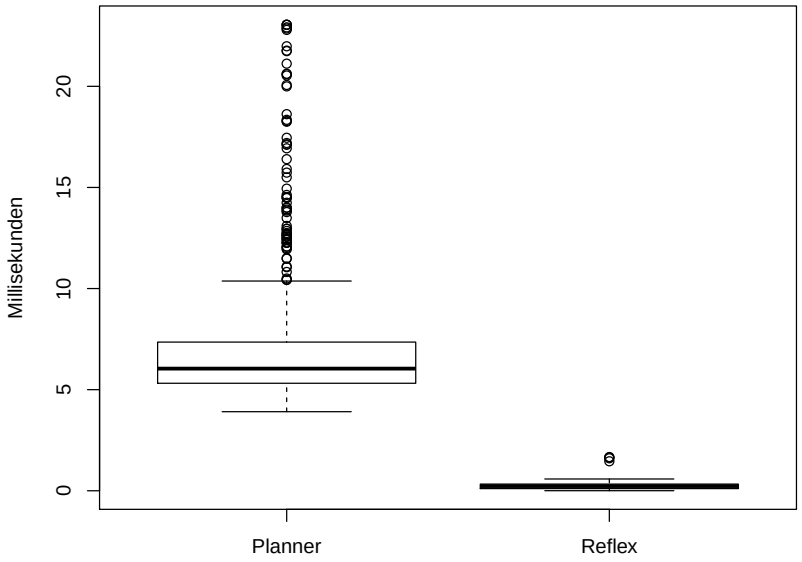


Abbildung 5.10: CCS1: Vergleich Planner Manager und Reflex Manager Zoom

Wie die Evaluierungsergebnisse zeigen, kann OCμ durch den Organic Manager im vorgestellten Cloud-Computing-Szenario die Selbst-X Eigenschaften umsetzen und zusätzlich die Ziele der Anwender erfüllen. Der Reflex Manager hat eine wesentlich kürzere Reaktionszeit als der Planer, kann jedoch nur auf Situationen reagieren die in ähnlicher Form bereits aufgetreten sind.

6 Erweitertes Modell

Im Regelfall wird ein Cloud-Computing-System von mehreren Personen genutzt. Die verschiedenen Entwickler können für das vorliegende Cloud-Computing-System Services programmieren und so Funktionalitäten bereit stellen. Andere Entwickler möchten solche Services nutzen und eigene Services darauf aufbauen. Das folgende Modell kann deshalb Abhängigkeiten zwischen Services abbilden.

So möchte beispielsweise ein Anwender einen Web Shop als Service innerhalb eines Cloud-Computing-Systems starten. Der Web Shop benötigt Zugriff auf einen Service, der Kundendaten verifiziert und die Bezahlung abwickelt. Dieser Kundendaten Service verfügt über die Fähigkeit eine gewisse Anzahl WebShop Services zu bedienen, kann aber redundant ausgeführt werden und so mehr Leistung zur Verfügung stellen. Abbildung 6.1 zeigt einen Kundendaten Service, der bis zu zehn Web Shop Services bedienen kann.

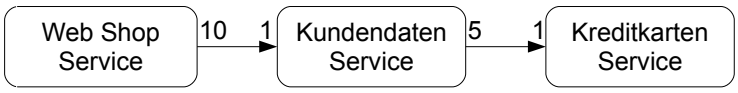


Abbildung 6.1: Beispiel: Abhängigkeiten zwischen Services

Der Kundenservice seinerseits baut wiederum auf einen Kreditkarten-Service auf, welcher ebenfalls nur einer begrenzten Anzahl anderer Services dienen kann. Durch mehrfache Ausführung des Kreditkarten-Services erhöht sich diese Anzahl jedoch. Abbildung 6.1 zeigt die Abhängigkeiten zwischen den Services. In diesem Beispiel kann ein Kreditkarten Service seine Dienste bis zu fünf Kundendaten Services zur Verfügung stellen.

Um solche Abhängigkeiten darstellen zu können, ist eine Erweiterung des Numerischen Modells notwendig. Zudem wurden Erfahrungen aus dem vorhergehenden Modell eingearbeitet und Optimierungen vorgenommen.

6.1 PDDL Modell

Wie im Numerischen Modell aus Abschnitt 4.1.3 gibt es in diesem Modell Services, deren gewünschte Anzahl laufender Instanzen vom Nutzer vorgegeben werden.

Zudem können diese Services von weiteren Services abhängig sein. Die nötige Anzahl dieser Services soll der Planer selbständig bestimmen.

6.1.1 Erweiterungen

Das Numerische Modell enthält bereits Informationen zu den verschiedenen Services. Jeder Service erhält als Erweiterung eine Kapazität, die der Nutzer oder der Service selbst angibt. Sie beschreibt wie vielen Instanzen eines anderen Services dieser Service bedienen kann. Des Weiteren wird die Summe der noch freien Kapazität als Wert berücksichtigt. Solange dieser Wert positiv ist, können weiter abhängige Services gestartet werden, ohne dass eine neue Instanz des zu Grunde liegenden Services gestartet werden muss. Um die Beziehung zwischen zwei Services zu modellieren, wird ein Boolesches PDDL Prädikat verwendet. Services, deren Anzahl der Planer selbständig festlegen soll, sind durch ein weiteres Boolesches Prädikat gekennzeichnet.

Das Modell enthält eine neue Aktion um die Zielanzahl an Services zu erhöhen und eine um sie zu verringern. Dies betrifft Services, von denen andere Services abhängig sind. Des Weiteren erhalten die `start` und `stop` Aktionen zusätzliche Vor- und Nachbedingungen. Die freie Kapazität der Services wird nach dem starten bzw. stoppen eines Services entsprechend erhöht oder verringert. Die `start` Aktionen erhalten zusätzlich als Vorbedingung, dass genügend freie Kapazität des zu Grunde liegenden Services vorhanden sein muss, um einen Service zu starten. Dadurch wird immer erst der zu Grunde liegende Service gestartet und somit Kapazität bereitgestellt, um anschließend den aufbauenden Service zu starten.

Bei den `stop` Aktionen kann dieses Verfahren nicht angewendet werden. Beim Stoppen eines abhängigen Services soll erst der abhängige Service gestoppt werden. Erst anschließend wird getestet, ob der zu Grunde liegende Service gestoppt werden sollte. Wenn die aktuelle Anzahl an abhängigen Services auch mit einem zu Grunde liegenden Service weniger bedient werden kann, wird ein zu Grunde liegender Service gestoppt. Dieses Verhalten ist mit Hilfe einer neuen Aktion implementiert, die testet, ob wenig genug Services laufen. Als Effekt wird ein Boolesches Prädikat auf wahr gesetzt, das für zu Grunde liegende Services als Ziel verwendet wird.

Die Knoten sind immer noch egoistisch modelliert. Jeder Knoten achtet darauf, nicht mehr Last zu haben als der Durchschnitt der anderen Knoten. Hat er mehr Last, kann er Services verschieben. Hat er weniger Last kann er aber keine Services aktiv an sich ziehen, sondern wartet bis seine Nachbarn Services zu ihm verschieben.

6.1.2 Optimierungen

Die Einführung von Abhängigkeiten erhöht die Komplexität und die Laufzeit des Planungsprozesses. Durch verschiedene Optimierungen soll die Komplexität des Planungsproblems verringert und so die Laufzeit des Planers verkürzt werden.

Dieses Modell enthält nur noch Aktionen für verschiebbare Services. Zum einen sind nicht verschiebbare Services in den hier vorgestellten Cloud- Computing-Szenarien nicht notwendig. Zum anderen kann so die, durch die eben beschriebenen Erweiterungen, erhöhte Komplexität verringert werden.

Services, für die der lokale Planer keine Ziele hat und die nicht lokal laufen, werden bei der Erstellung des PDDL Problems nicht betrachtet. Es ist nicht notwendig diese Services mit einzubeziehen, da der lokale Planer keine Aktionen hat, die auf solche Services anwendbar sind. Services, für die der lokale Planer keine Ziele hat, die jedoch lokal laufen, werden zusammengefasst und im Modell als ein einziger Service berücksichtigt. Diese Zusammenfassung ist möglich, da der Planer solche Services nur verschieben kann und sonst keine weiteren Aktionen anwendbar sind. Welche Services dieser Gruppe angehören wird abgespeichert, so dass später bei der Ausführung der Aktion ein bestimmter Service aus dieser Gruppe ausgewählt wird.

6.2 Vergleich der Modelle

Tabelle 6.1 zeigt die Unterschiede zwischen den Modellen. Das erweiterte Modell berücksichtigt Abhängigkeiten zwischen Services, während das Numerische Modell nicht über diese Fähigkeit verfügt. Trotz einiger Optimierungen hat das erweiterte Modell jedoch eine höhere Komplexität als das Numerische Modell. Das erweiterte Modell verfügt über einen Boolschen Wert und zwei numerische Werte mehr, als das Numerische Modell. Zudem hat es eine Aktion mehr als das Numerische Modell.

	Numerisches Modell	Erweitertes Modell
Service Abhängigkeit	Nein	Ja
PDDL Model		
:types	3	3
:predicates	4	5
:functions	8	10
:actions	10	11

Tabelle 6.1: Vergleich: Numerische vs. erweitertes Modell

Das bereits aus Abschnitt 5.2 bekannte Szenario mit drei Nutzern und zehn Knoten wird mit dem erweiterten Modell erneut evaluiert. Da in diesem Szenario keine Abhängigkeiten zwischen den Services existieren, überwiegen die Effekte der Optimierungen.

In Bezug auf Erfüllung der Anwenderziele, welche in Abbildung 5.1 dargestellt sind, und der Selbst-Heilung unterscheiden sich die Ergebnisse beider Modelle nicht wesentlich. Auch das erweiterte Modell erfüllt die Ziele aller drei Anwender und startet ausgefallene Services neu. Wie Abbildung 6.2 zeigt, werden die Services gleichmäßig auf verschiedene Knoten verteilt.

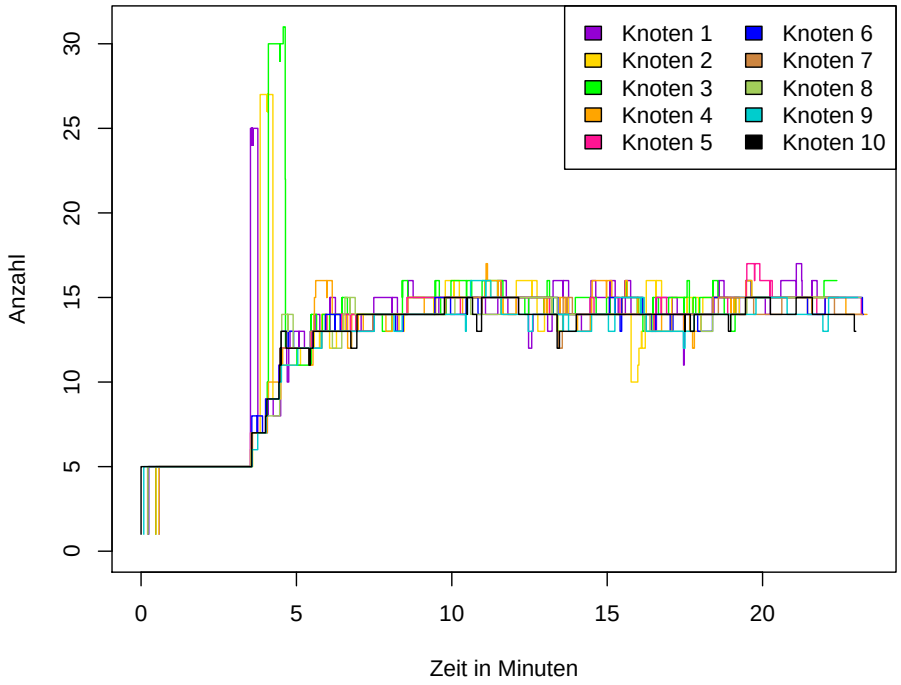


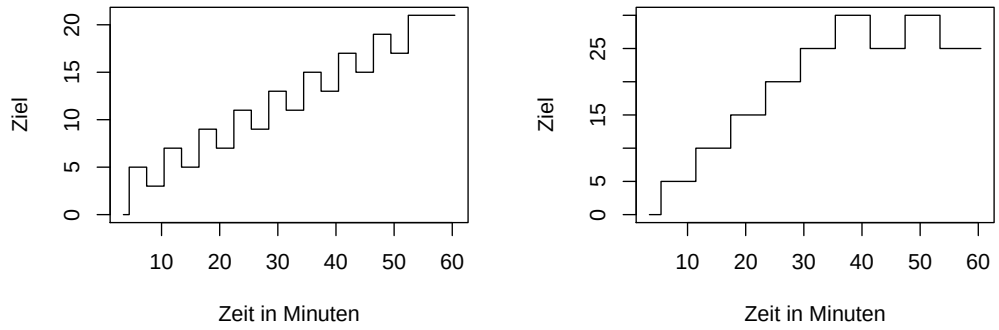
Abbildung 6.2: Summe Services je Knoten

Im Gegensatz zum ersten Modell, verringert sich die Planungszeit durch die Optimierungen deutlich. Die durchschnittliche Planungszeit sank von 33,85 Millisekunden auf 10,54 Millisekunden. Die durchschnittliche Planlänge aller Pläne unterscheidet sich bei beiden Modellen nur um 3%. Die Verringerung der Erstelldauer ist vor allem auf die eingesetzten Optimierungen zurückzuführen. Durch den Ausschluss bzw. das Zusammenfassen von Services enthalten die PDDL-Probleme des neuen Modells weniger Objekte, als die des alten Modells. Durch die geringere Anzahl an Objekten sind pro Planungsschritt weniger Aktionen anwendbar, die der Planer testen kann. Dadurch verringert sich der Aufwand, einen Plan zu finden.

6.3 Cloud-Computing-Szenario mit Service-Abhängigkeiten

An diesem Cloud-Computing-Szenario (CCS2) nehmen zehn Knoten teil. Es gibt insgesamt drei simulierte Anwender, die den jeweiligen Planern auf ihren Knoten Ziele vorgeben. Dieses Szenario erstreckt sich über 60 Minuten. Abgesehen von den Zielen der Anwender und der Dauer des Szenarios werden die übrigen Parameter von OCμ genauso gewählt wie bei CCS1 aus Kapitel 5.2.

Der erste Anwender verfolgt eine Strategie mit einer tendenziell steigenden Anzahl Services. Dies soll ein sich steigernder Bedarf an Services simulieren, wobei der Bedarf jedoch nicht nur steigt, sondern zwischendurch auch wieder sinkt. Wie in Abbildung 6.3.a zu sehen ist, erhöht der erste Anwender die gewünschte Anzahl an Services A1 regelmäßig um vier und verringert sie im darauf folgenden Schritt um zwei bis die Gesamtanzahl Services A1 21 erreicht hat. Der zweite Anwender



(6.3.a) Ziel Service A1 Anwender 1

(6.3.b) Ziel Service A2 Anwender 2

Abbildung 6.3: CCS2: Ziele verschiedener Anwender

erhöht die Anzahl an Services erst, um dann später eine periodisch schwankende Anzahl an Services zu fordern, wie in Abbildung 6.3.b dargestellt. Er erhöht die gewünschte Anzahl an Services in gleichmäßigen Abständen bis 30 Services A2. Anschließend schwankt der Zielwert zwischen 30 und 25. Dies soll periodische wiederkehrendes Interesse simulieren.

Der dritte Anwender verfolgt die in Abbildung 6.4 dargestellte Strategie. Er steigert die Anzahl an Services schnell, um dann eine tendenziell fallende Anzahl an Services festzulegen. Dabei fällt der Bedarf an Services nicht kontinuierlich, sondern steigt zwischendurch leicht an.

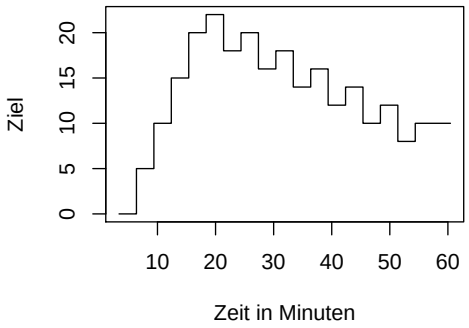


Abbildung 6.4: CCS2: Ziel Service A3 Anwender 3

Jeder der Anwender Services ist abhängig von anderen Services. Abbildung 6.5 stellt die Abhängigkeit von Service A1 dar. Service A1 nimmt die Dienste von Service B1 in Anspruch. Dabei kann eine Instanz des Services B1 bis zu fünf Instanzen des Services A1 versorgen.

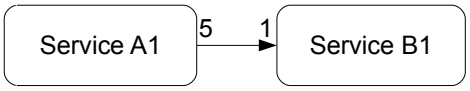


Abbildung 6.5: CCS2: Abhängigkeiten Service A1 Anwender 1

Service A2 ist von Service B2 abhängig, der wiederum von Service C2 abhängig ist wie aus Abbildung 6.6 ersichtlich ist. Ein Service C2 kann von bis zu drei Services B2 genutzt werden. Jede Service-Instanz von B2 stellt seine Fähigkeiten bis zu zehn Services A2 zur Verfügung. Das heißt, dass ein Service C2 indirekt bis zu 30 Services A2 bedienen kann, bevor ein weiterer Service C2 gestartet werden muss.

Abbildung 6.6: CCS2: Abhängigkeiten Service A2 Anwender 2

Abbildung 6.7 zeigt die Abhängigkeiten des Services A3. Service A3 benötigt den Service B3, welcher wiederum auf Service C3 basiert. Ein Service C3 kann bis zu fünf Services B3 versorgen. Ein Service B3 kann seine Dienste wiederum bis zu fünf Services A3 zur Verfügung stellen. Ein Service C3 kann bei fünf laufenden Services B3 seine Dienste somit bis zu 25 Services A2 zur Verfügung stellen.

Abbildung 6.7: CCS2: Abhängigkeiten Service A3 Anwender 3

6.4 Evaluierungsergebnisse erweitertes Szenario

Im Folgenden werden zwei Evaluierungen näher betrachtet. In der ersten Evaluierung fallen keine Services aus. In der zweiten Evaluierung fallen häufig Services aus. Beide Evaluierungen werden anhand der Evaluierungskriterien aus Abschnitt 5.1 analysiert.

6.4.1 Ohne ausfallende Services

Um die Selbst-Optimierung umzusetzen, soll der Planer eine gleichmäßige Auslastung der Knoten erreichen. Durch häufiges Starten und Stoppen von Services entsteht eine gewisse Varianz. Abbildung 6.8 zeigt, dass sich die Auslastung der Knoten trotzdem schnell einander annähert. Die Ausreißer nach unten resultieren aus dem egoistischen Verhalten der Knoten. Auf diesen Knoten wurden Services gestoppt, sie können jedoch keine Service von anderen Knoten zu sich verschieben. Stattdessen teilen die Knoten ihre neue Auslastung den anderen Knoten mit und erhalten von ihnen Services.

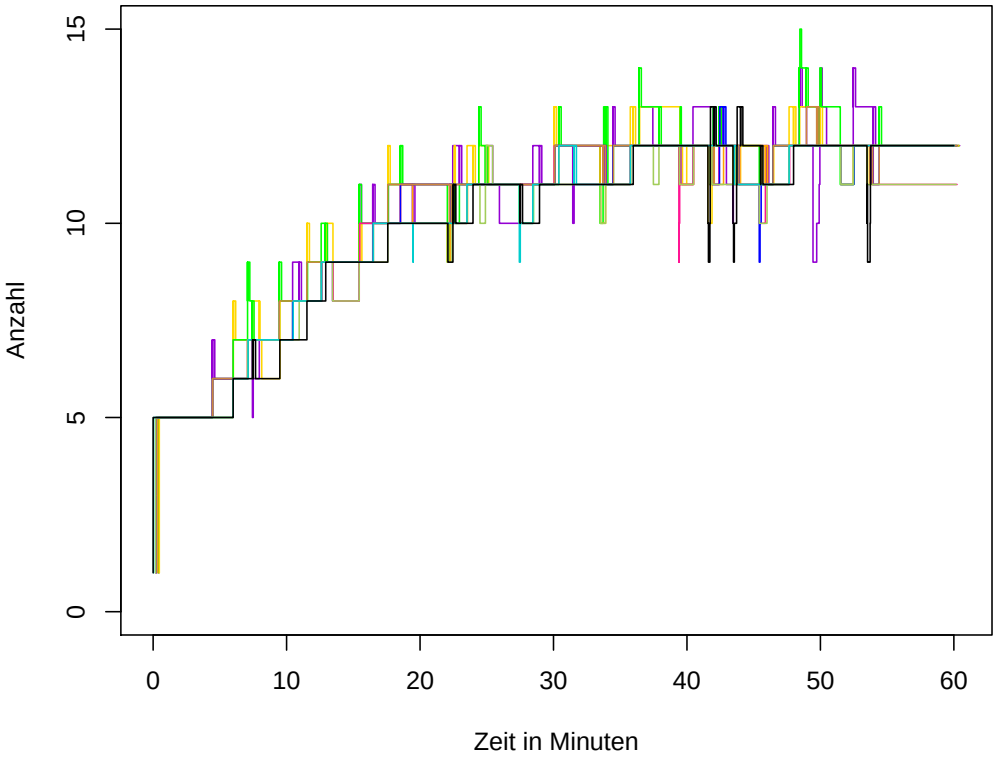


Abbildung 6.8: CCS2: Summe Services je Knoten

Ziele der Anwender

Zu Beginn der Evaluierung erhöht sich die Gesamtanzahl Services ständig. Dies liegt daran, dass alle drei Anwender anfangs ihre Ziele steigern. Nachdem der zweite Anwender beginnt seine Ziele in periodische Schwankungen zu ändern, stabilisiert sich die Anzahl an Services. Die Ziel-Änderungen des ersten und dritten Anwenders gleichen sich in der Summe fast wieder aus.

Gegen Ende der Evaluierung, nachdem alle Anwender ihr letzte Ziel-Änderung vorgenommen haben, hat jeder Knoten eine Last von 11 oder 12 Services. Dabei werden keine Service mehr verschoben.

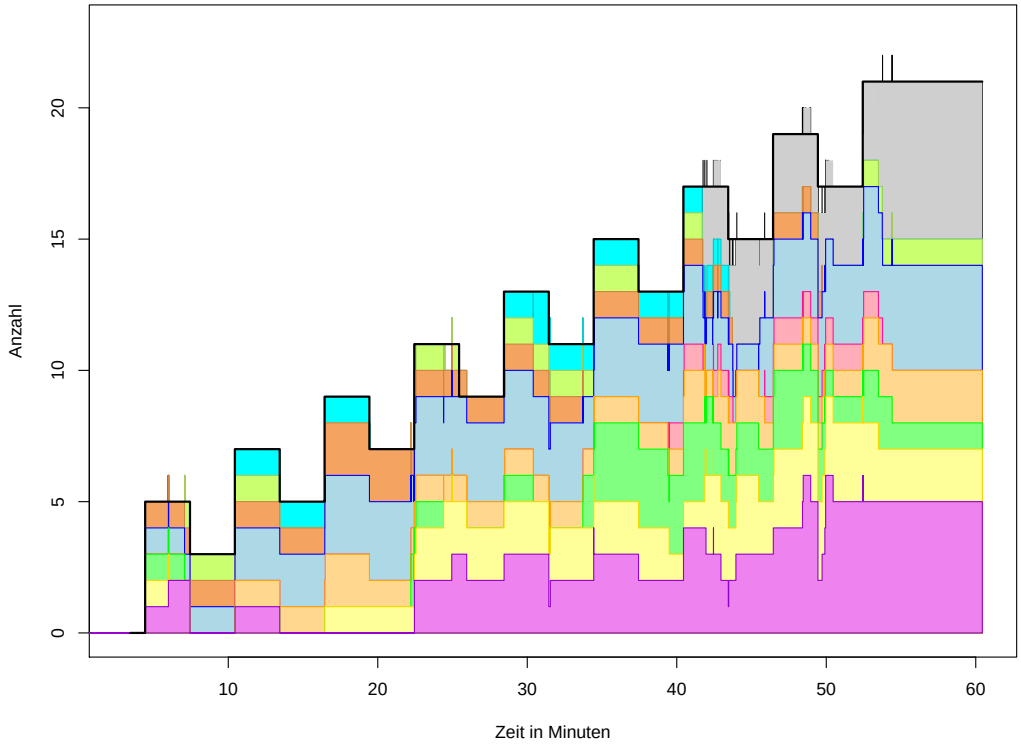


Abbildung 6.9: CCS2: Summe aller Services A1

In Abbildung 6.9 sind die Ziele des ersten Anwenders als schwarze Linie eingetragen. Die unterschiedlichen Farben symbolisieren die verschiedenen Knoten, auf denen Instanzen des Services A1 laufen. Wie man sieht, werden die Services auf verschiedene Knoten verteilt und der Planer erfüllt die Wünsche des Anwenders sehr gut.

Service A1 ist von Service B1 abhängig. Wie bereits beschrieben kann ein Service B1 bis zu fünf Services A1 bedienen. In Abbildung 6.10 entspricht die schwarze Linie der notwendigen Anzahl an Services B1, also die Anzahl an Services A1 geteilt durch fünf aufgerundet auf die nächste Ganzzahl. Wie man sehen kann, erreicht

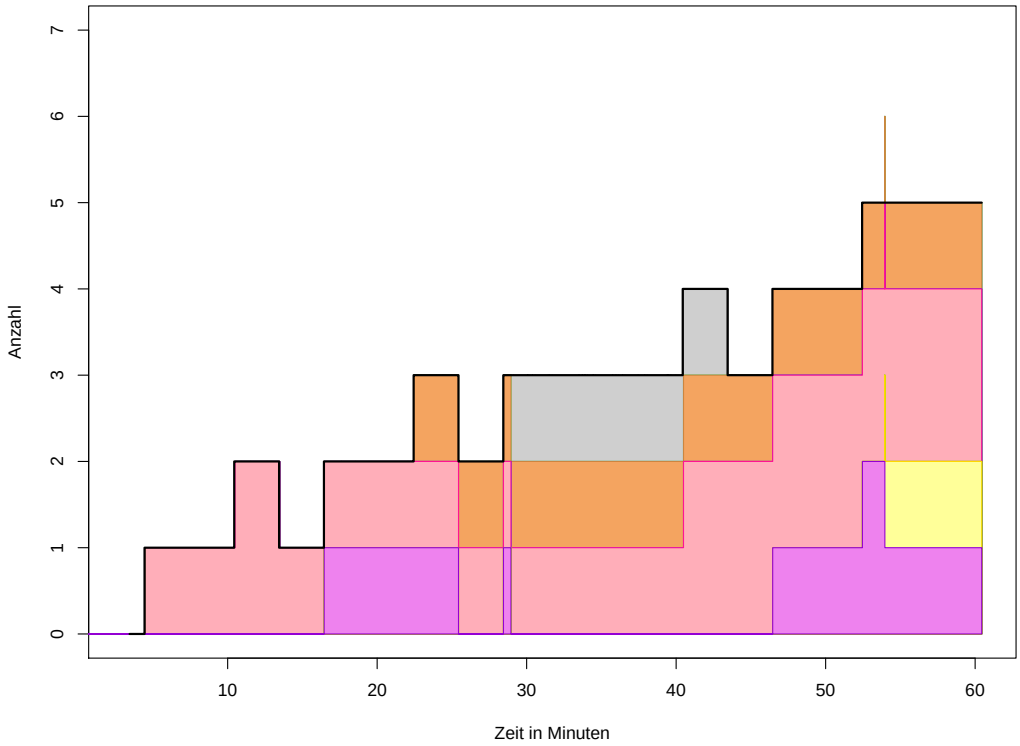
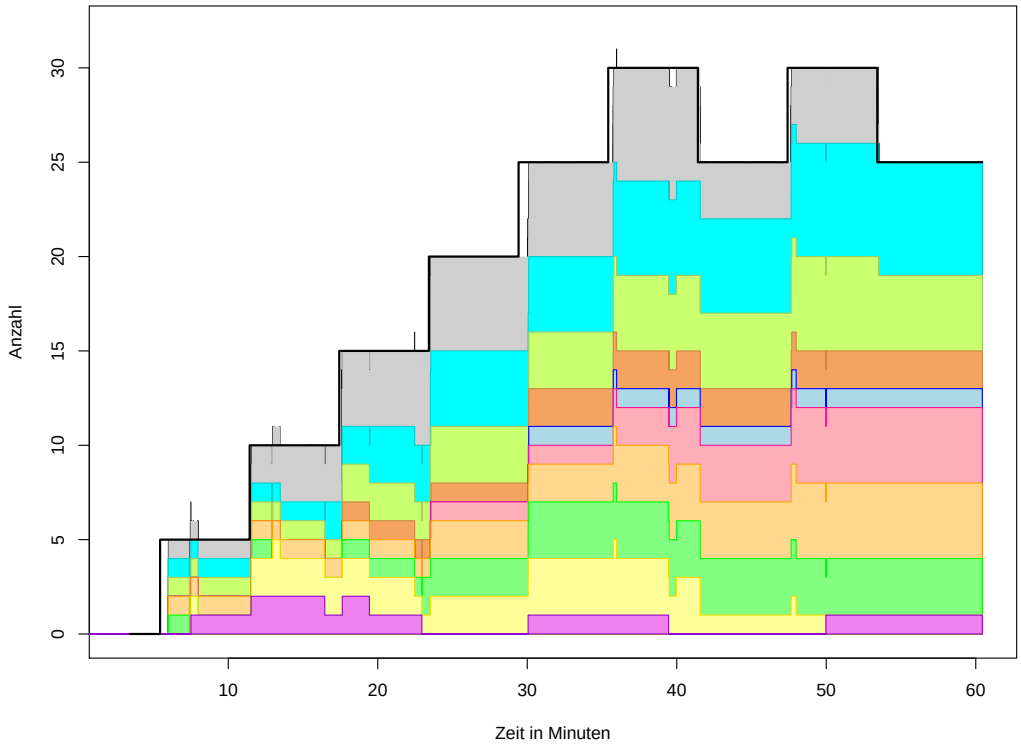


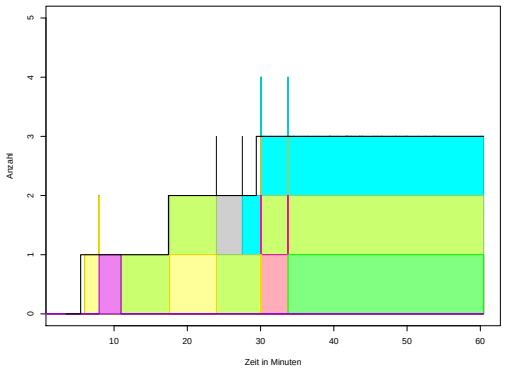
Abbildung 6.10: CCS2: Summe aller Services B1

der Planer die optimale Anzahl an Services B1. Es werden weder zu wenig, noch zu viele gestartet. Die unterschiedlichen Farben zeigen wieder die Verteilung der Services auf verschiedene Knoten. Da auch noch andere Services im Netz auf die Knoten verteilt werden, erhält nicht jeder Knoten die gleiche Anzahl an Services B1.

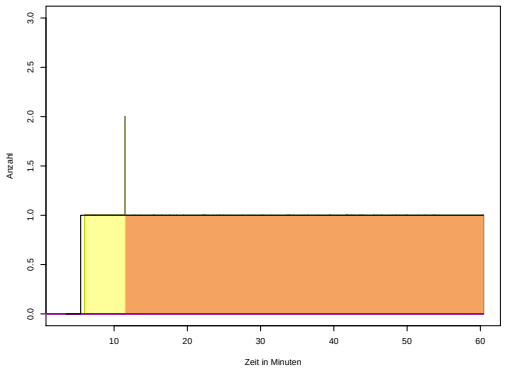
Die Gesamtanzahl Services A2, auf die sich die Ziele des zweiten Anwenders beziehen, sind in Abbildung 6.11.a dargestellt. Die Zielanzahl des Anwenders wird erreicht und die Services wie gewünscht auf verschiedene Knoten verteilt.



(6.11.a) Summe Services A2



(6.11.b) Summe Services B2



(6.11.c) Summe Services C2

Abbildung 6.11: CCS2: Summe verschiedener Services im Zeitverlauf

Da zu Beginn die gewünschte Anzahl an Services A2 ständig steigt, steigt auch die benötigte Anzahl an Services B2. Ein Service B2 kann bis zu zehn Services A2 dienen. Insgesamt laufen maximal 30 Services A2, somit werden bis zu drei Services B2 benötigt, deren Summe in Abbildung 6.11.b angetragen ist. Diese drei Ser-

vices B2 reichen für 20 bis 30 Services A2 aus, ohne dass ein Service B2 gestoppt oder gestartet werden muss. Ein Service C2 kann bis zu drei Services B2 versorgen. Da höchstens drei Services B2 benötigt werden, reicht in diesem Fall ein Service C2 aus. Abbildung 6.11.c zeigt, dass der Planer diesen einen Service C2 startet und einmal verschiebt.

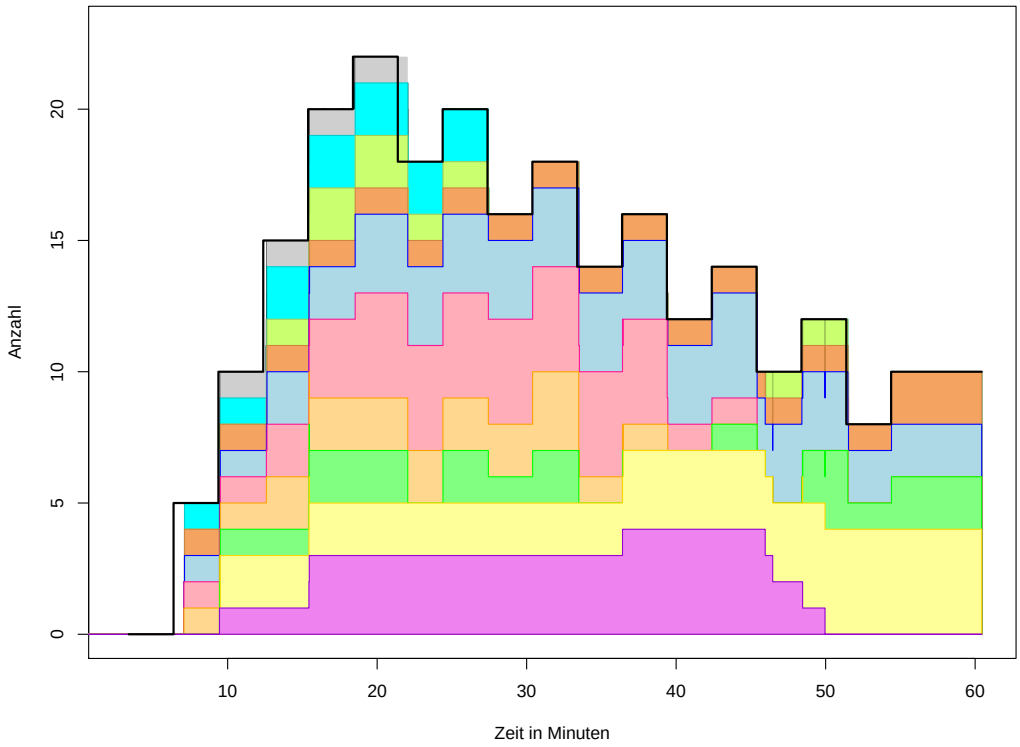


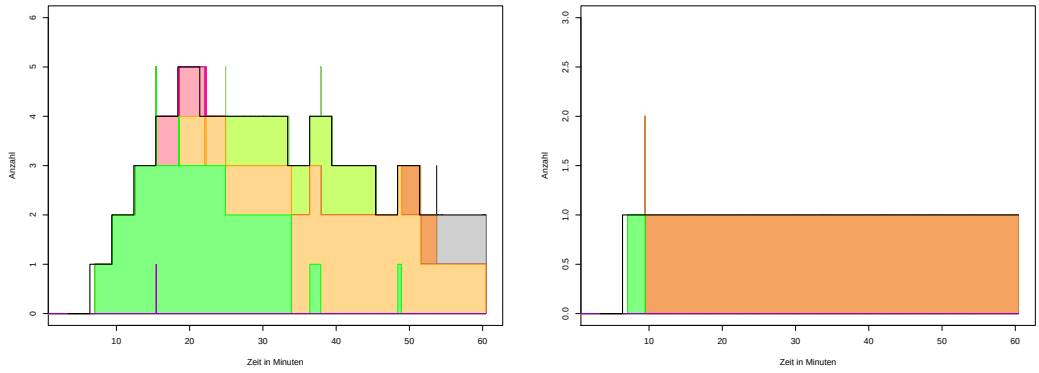
Abbildung 6.12: CCS2: Summe aller Services A3

Die gewünschte Anzahl an Services A3 des dritten Anwenders ist in Abbildung 6.12 als schwarze Linie eingetragen. Dabei kann man beobachten, dass es zu Beginn eine kurze Lücke zwischen dem Hinzufügen des Ziels und der Reaktion des Systems gibt. In diesem Fall war der Planer kurz vor dem Hinzufügen des Zieles aktiv und wartet, bis er das nächste Mal regulär angestoßen wird. Dieses Verhalten tritt noch einmal in der 22. Minute auf, wenn der Anwender das Ziel von 22 Services A3 auf 18 reduziert. Die Services werden auf verschiedene Knoten verteilt.

Wie bereits beschrieben, ist der Service A3 von Service B3 abhängig. Ein Service B3 kann seine Dienste bis zu fünf anderen Services zur Verfügung stellen. Da zu Beginn die Services A3 in fünfer Schritten erhöht werden, steigt hier auch die nötige Anzahl an Services B3 je um 1, bis zu einem Maximum von fünf Services B3,

wie in Abbildung 6.13.a erkennbar. Nachdem der Anwender die gewünschte Anzahl an Services A3 verringert, verringert sich auch die Anzahl an Services B3. Vier Services B3 können zwischen 15 und 20 Services A3 bedienen. Da sich die Ziele des Anwenders zwischen Minute 22 und 33 innerhalb dieser Grenzen bewegen, verändert sich die Anzahl an benötigten Services B3 während dieser Zeit nicht.

Ein Service C3 kann bis zu fünf Services B3 versorgen, deshalb ist hier nur eine Instanz dieses Services von Nöten. Wie Abbildung 6.13.b zeigt, wird dieser Service kurz nach dem Starten verschoben.



(6.13.a) CCS2: Summe Services B3 (6.13.b) CCS2: Summe Services C3

Abbildung 6.13: CCS2: Summe aller Service B3 und C3

Reflex und Planner Manager

In dieser Evaluierung ergibt sich eine relativ hohe Trefferrate der Reflex Base. Wie in Abbildung 6.14 erkennbar ist, haben alle Knoten eine Trefferrate von über 30%. Da in dieser Evaluierung keine Services ausfallen, variieren die Zustände nicht so stark, wie bei der vorherigen Evaluierung. Somit werden öfters Zustände als ähnlich erkannt und die Trefferrate erhöht sich. Der Actuator hat in dieser Evaluierung bei seinem Vergleich der gewählten Reflexe mit dem optimalen Plan keine Abweichungen festgestellt. Obwohl auf Knoten eins bis drei die Anwender ständig die Ziele des Planers ändern und somit die Reflex Base gelöscht wird, ist die Trefferrate nicht niedriger als bei den Knoten 9 und 10. Knoten 9 und 10 haben jeweils ca. zehn Treffer weniger als die Knoten vier bis acht. Diese fehlenden Treffer sind zeitlich zufällig verteilt.

In Abbildung 6.15.a und 6.15.b wird die Zeit dargestellt, bis der Planner bzw. die Reflex Base einen Plan erstellt oder ausgesucht haben. Wie man erkennen kann, gibt es einige Pläne, für die der Planner ca. 40 Sekunden benötigt. Die meisten Pläne jedoch benötigten deutlich weniger Zeit. Der Reflex Manager hingegen ist im

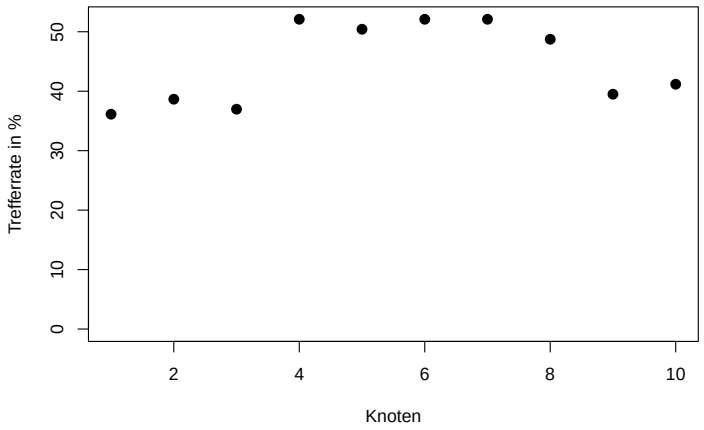
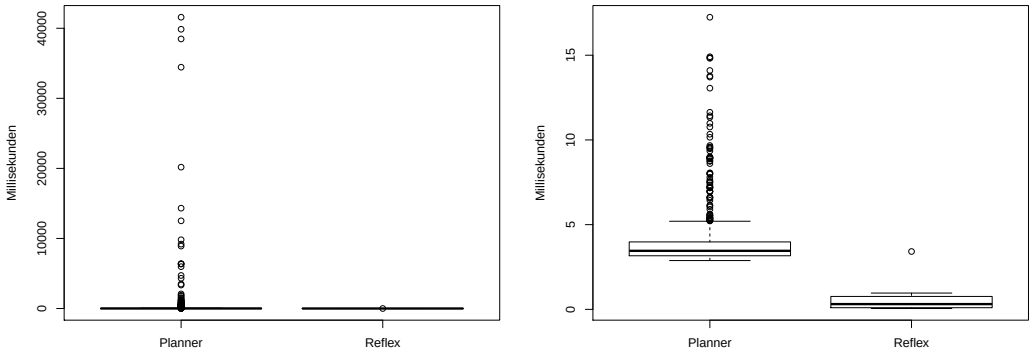


Abbildung 6.14: CCS2: Trefferrate der Reflex Base

Regelfall deutlich schneller, als der Planner Manager, wie man in der vergrößerten Abbildung 6.15.b erkennen kann. Lediglich einmal wurde eine höhere Zeitspanne gemessen, jedoch war dabei der Reflex Manager immer noch schneller, als der zugehörige Planer.



(6.15.a) Vergleich Planner Manager Reflex Manager

(6.15.b) Zoom: Vergleich Planner Manager Reflex Manager

Abbildung 6.15: CCS2: Vergleich Reflex vs. Planner Manager

Die vorliegende Evaluierung zeigt, dass der Organic Manager mit dem erweiterten Modell das System im Regelfall schnell und zuverlässig steuern kann. Die Ziele der Anwender werden erfüllt und die Knoten gleichmäßig ausgelastet. Der Reflex Manager beschleunigt das System, indem er Pläne zwischenspeichert.

6.4.2 Mit ausfallende Services

Dieser Evaluierung wird das Szenario aus Abschnitt 6.3 zu Grunde gelegt, jedoch wird das System stark durch ausfallende Services gestört. Dabei ist es Zufall, welcher Service wann ausfällt. Auf jedem Knoten ist ein Programm aktiv, dass in regelmäßigen Abständen entscheidet ob, und welcher Service ausfällt.

Selbst-Optimierung

Durch die häufigen Ausfälle erhöhen sich die Schwankungen der Last der einzelnen Knoten. Dennoch wird eine gleichmäßige Auslastung der Knoten erreicht, wie man in Abbildung 6.16 sehen kann. Der Unterschied zu Abbildung 6.9 liegt lediglich in der größeren Varianz der hier gemessenen Werte. Extreme Ausreißer gibt es keine, einzelne Ausreißer werden wie bereits bei der vorherigen Evaluierung schnell korrigiert. Da die Ziele der Anwender die gleichen wie in der Evaluierung aus dem vorhergehenden Abschnitt sind, ergibt sich auch hier eine zu Beginn steigende und später gleichbleibende Anzahl an Services.

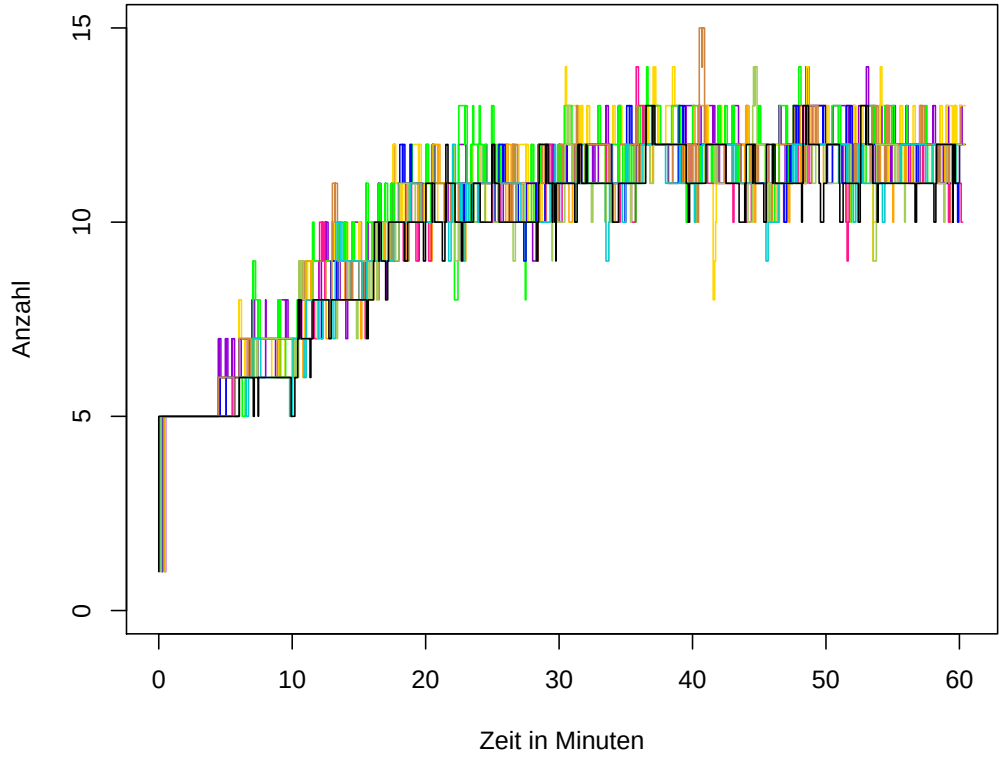


Abbildung 6.16: CCS2: Summe Services je Knoten mit Service-Ausfällen

Die Ziele des ersten Anwenders sind in Abbildung 6.17 wieder als schwarze Linie eingetragen. Wenn der Service A1 auf irgendeinem Knoten ausfällt, wird das mit Hilfe eines kleinen Kreuzes am oberen Rand der Abbildung markiert. Trotz der häufigen Ausfälle kann der Planer die Vorgaben des Anwenders einhalten. Wie man bei Minute 22 sehen kann, übersteuert der Planer einmal kurz. In diesem Fall

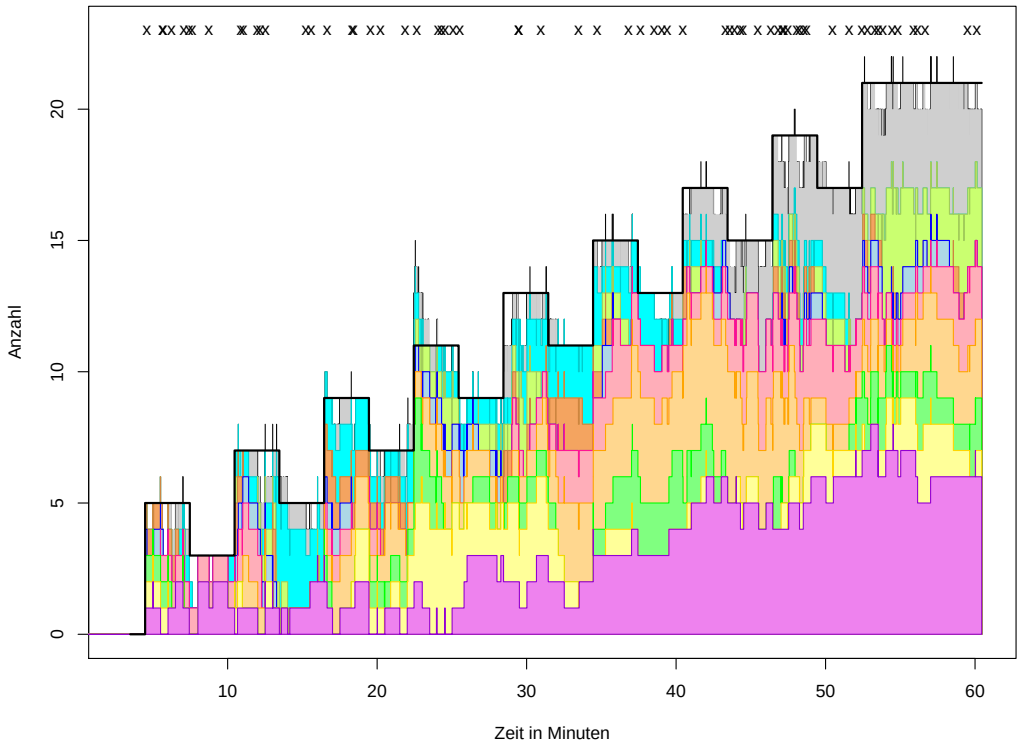


Abbildung 6.17: CCS2: Summe aller Services A1

lagen dem Planer falsche Informationen vor. Laut seiner Daten liefen zwei Services A1 weniger, als tatsächlich im Netz liefen. Dies kann in sehr seltenen Fällen vorkommen. Dabei weist der Actuator einen anderen Knoten an, einen Service neu zu starten oder versucht einen Service zu verschieben und vermerkt dies in der Fact Base. Bevor der andere Knoten jedoch das Starten des Services abgeschlossen hat, wird eine Nachricht verschickt in dem dieser neue Service noch nicht enthalten ist. Der erste Knoten übernimmt diese Information in die Fact Base. Wenn in dieser Zeit geplant wird, erhält der Planer veraltete Daten. Sobald das Starten des Services auf dem anderen Knoten abgeschlossen ist und erneut eine Nachricht gesendet wird, korrigiert sich die Fact Base. Der Planer erhält dann den korrekten Zustand und kann einen entsprechenden Plan erstellen.

Im späteren Verlauf häufen sich die Ausfälle besonders zwischen Minute 40 und 50. Trotz dieser extremen Situation schafft der Planer es die Zielanzahl zu halten.

Abbildung 6.18 zeigt die Anzahl an Services B1, wobei ein Service B1 bis zu fünf Services A1 bedienen kann. Wie man sieht fällt Service B1 wesentlich seltener aus, da nicht so viele Instanzen von ihm laufen. Auch Service B1 wird auf verschiedene Knoten verteilt. Der Planer schafft es, die optimale Anzahl an Services B1 bereitzustellen. Die einzelnen Striche nach oben werden von Verschiebungen von Services zwischen den Knoten verursacht.

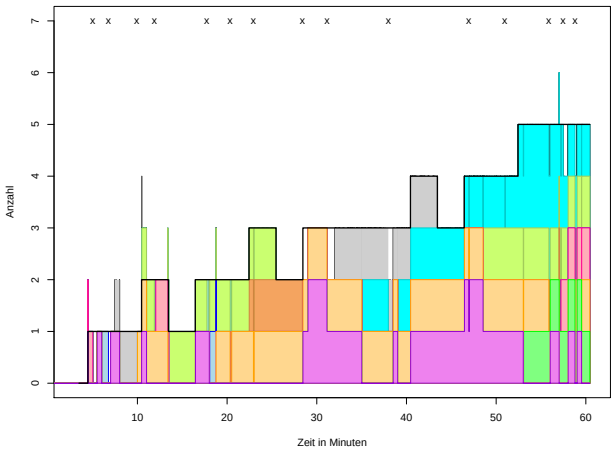


Abbildung 6.18: CCS2: Summe aller Services B1

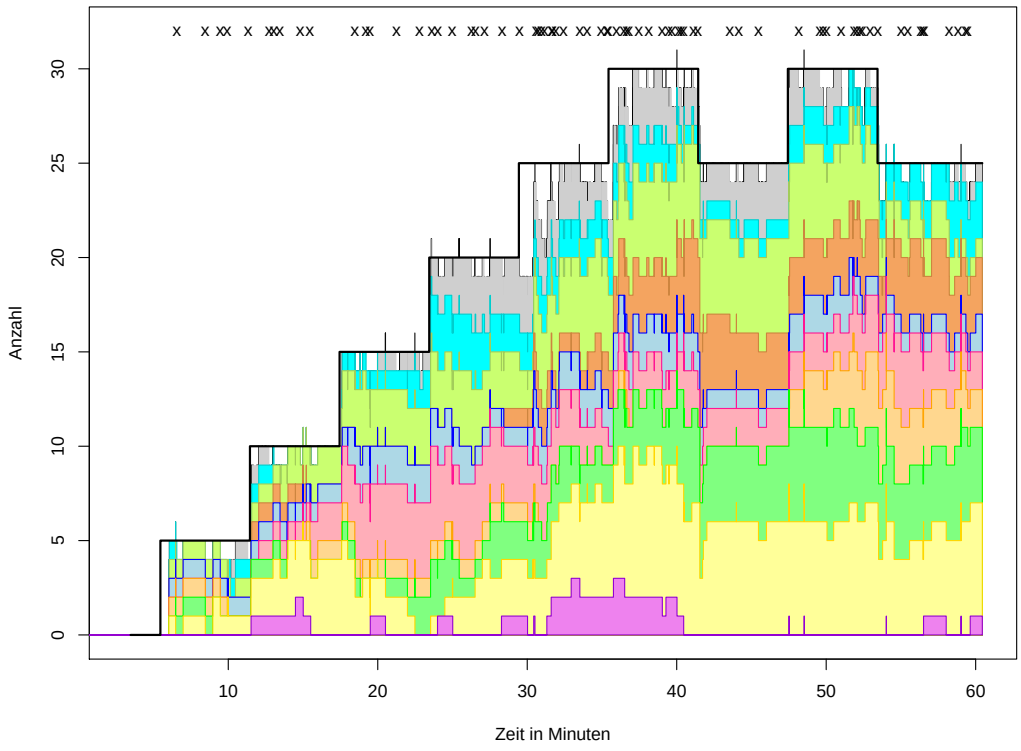


Abbildung 6.19: CCS2: Summe aller Services A2

Die Gesamtanzahl aktiver Instanzen des Services A2 ist in Abbildung 6.19 aufgetragen. Auffallend sind hierbei die kleinen Verzögerungen zu Beginn und bei Minute 30 der Evaluierung. Der Planer wurde in diesen Fällen nicht direkt nach Hinzufügen des Zieles angestoßen, sondern erst einige Sekunden später. Dadurch

entsteht eine sichtbare, kurze Verzögerung, bis die Ziele des Anwenders erfüllt werden. Trotz der häufigen Ausfälle um Minute 30 und 40 erreicht das System immer wieder die gewünschte Anzahl an Services.

Service B2, von dem A2 abhängt, wird ebenfalls gestartet, wie Abbildung 6.20 zeigt. Da drei Services B2 ihre Dienste bis zu 30 Services A2 zur Verfügung stellen können, werden nicht mehr gestartet. Wie man sehen kann, werden die Services auf unterschiedliche Knoten verteilt. Das es nur wenige Services B2 gibt, fallen sie auch nicht so häufig aus. Der Planer schafft es, jeden Ausfall zeitnah zu kompensieren. Wie in der vorangegangenen Evaluierung bereits gezeigt, ist nur ein Service C2 nötig, um die Abhängigkeiten von B2 zu erfüllen. Da es sich nur um einen Service handelt, wird hier auf diese Abbildungen verzichtet.

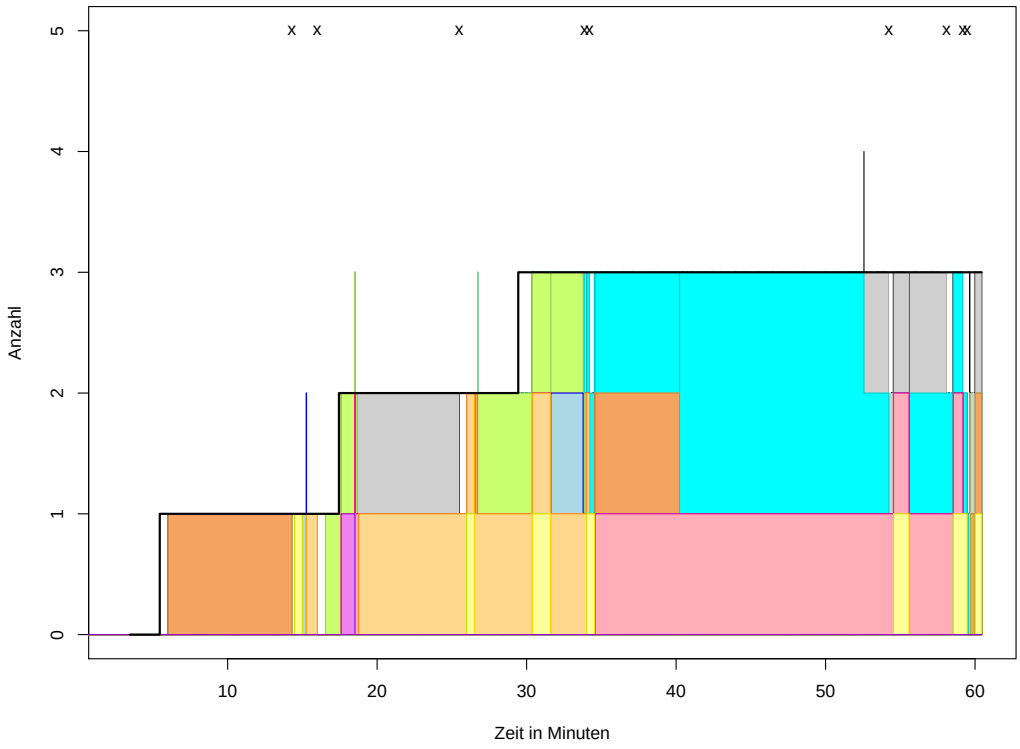


Abbildung 6.20: CCS2: Summe aller Services B2

Die Ziele des dritten Anwenders sind in Abbildung 6.21 als schwarze Linie gekennzeichnet. Wieder kann man zu Beginn einen Abstand zwischen dem Hinzufügen des Ziels und der Umsetzung erkennen. Dies liegt daran, dass der Planer

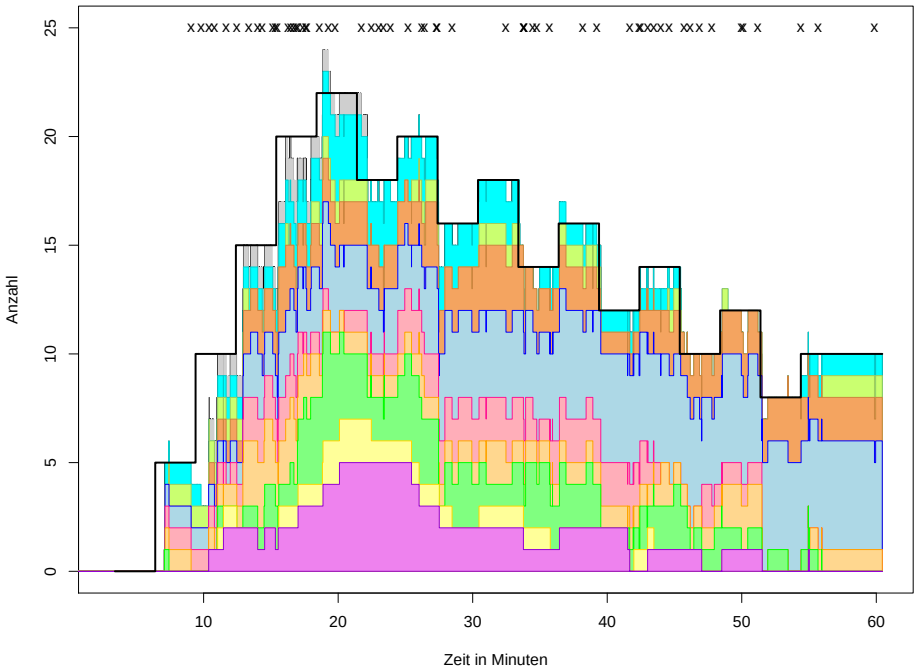


Abbildung 6.21: CCS2: Summe aller Services A3

periodisch angestoßen wird. Die Lücke nach der Ziel-Änderung von fünf auf zehn Services A3 liegt an der relativ hohen Erstelldauer dieses Planes. Dadurch entsteht eine zeitliche Verschiebung. Zudem fallen in dieser Zeit zwei Services aus, die der Planer im nächsten Schritt neu startet. Service A3 ist vor allem im ersten Drittel der Evaluierung Opfer häufiger Ausfälle. Trotz dieser extremen Störungen kann der Planer die Ziele des Anwenders oft erfüllen.

Die Gesamtanzahl an Services B3 ist in Abbildung 6.22 aufgetragen. Auch hier sind Ausfälle zu erkennen, welche der Planer jedoch ausgleicht. Auch die Instanzen des Services B3 werden auf verschiedene Knoten verteilt.

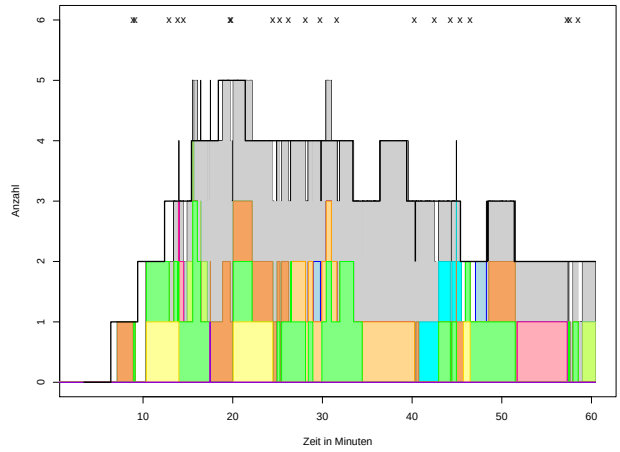


Abbildung 6.22: CCS2: Summe aller Services B3

Reflex und Planner Manager

Abbildung 6.23 zeigt die Trefferrate der Reflex Base. Auffällig ist, dass die Trefferrate deutlich niedriger ist, als in der vorherigen Evaluierung, bei der keine Services ausfielen. Durch die vielen Ausfälle schwanken auch die gemessen Zustände stark. Dadurch werden wesentlich weniger Zustände als ähnlich erkannt, als dies noch in der vorherigen Evaluierung der Fall war.

Da die Anzahl der Treffer pro Knoten nur zwischen ca. 7 und 11 liegt, kann über die unterschiedlichen Ergebnisse der Knoten keine Aussage getroffen werden.

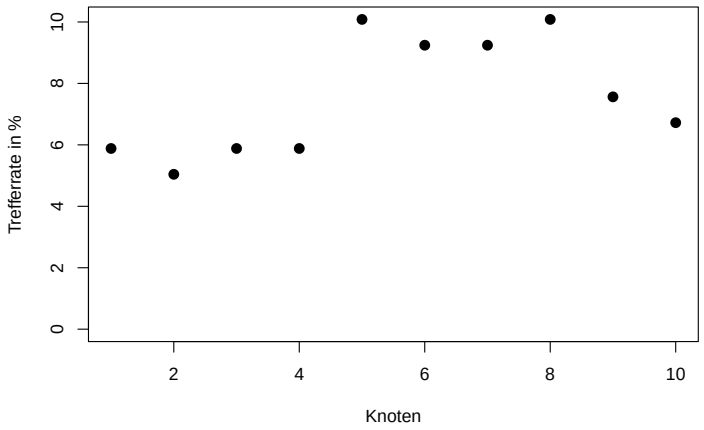
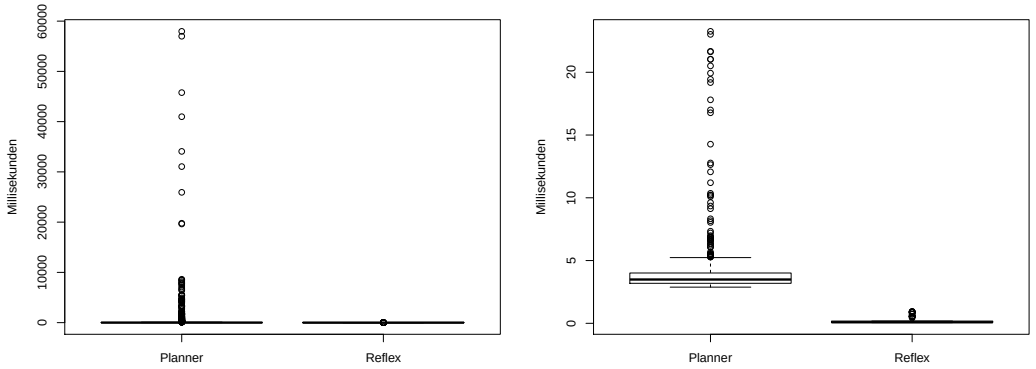


Abbildung 6.23: CCS2: Trefferrate der Reflex Base mit Service-Ausfällen



(6.24.a) Vergleich Planner Manager Reflex Manager

(6.24.b) Zoom: Vergleich Planner Manager Reflex Manager

Abbildung 6.24: CCS2: Vergleich Reflex Manager mit Planner Manager

Die maximale Erstelldauer eines Planes ist höher als in der vorherigen Evaluierung aus Abschnitt 6.4.1. Wie aus Abbildung 6.24.a erkennbar ist, gibt es drei Pläne, die eine höhere Erstelldauer haben als in der vorhergehenden Evaluierung. Beides Mal benötigten die meisten Pläne jedoch weniger als zehn Sekunden für ihre Erstellung. Abbildung 6.24.b zeigt, dass der Reflex Manager schneller einen Plan auswählen kann, als der Planner Manager Zeit für die Erstellung eines Planes benötigt.

Die durchschnittliche Erstelldauer für einen Plan beträgt in dieser Evaluierung 558 Millisekunden, während in der Evaluierung aus Abschnitt 6.4.1 durchschnittlich nur 295 Millisekunden für die Generierung eines Planes benötigt wurden. In beiden Evaluierungen beträgt die maximale Länge eines Planes 14 Aktionen. Jedoch ist die durchschnittliche Planlänge der vorherigen Evaluierung mit $2,38$ etwas niedriger als in der aktuellen Evaluierung mit $2,72$ Aktionen pro Plan. Das schlechtere Ergebnis dieser Evaluierung liegt vor allem an der höheren Varianz der Anzahl an Services durch die häufigen Service-Ausfälle. Dadurch ergibt sich für die Knoten in dieser Evaluierung ein höherer Planungsaufwand.

Wie diese Evaluierung zeigt, kann der Planer auch bei häufigen Ausfällen von Services die Selbst-X Eigenschaften noch erfolgreich umsetzen. Das System optimiert sich selbst und verteilt die Services gleichmäßig auf verschiedene Knoten. Die Ziele der Anwender werden erfüllt, jedoch können kleinere Fehler wie beispielsweise Übersteuerung oder eine leicht zeitversetzte Erfüllung der Ziele auftreten. Durch die Selbst-Heilung werden ausgefallene Services zeitnah wieder gestartet. Allerdings erhöht sich der Planungsaufwand durch häufige Ausfälle und die durchschnittliche Erstelldauer der Pläne steigt.

6.5 Verteilte Abhängigkeiten

In den vorhergehenden Szenarien sind die Services der Anwender jeweils disjunkt. Die Aufgaben der Planer sind in diesen Szenarien somit auch unabhängig voneinander. Im folgenden Szenario ist das Wissen über die Abhängigkeiten verteilt abgespeichert. Dies kann geschehen, wenn die Anwender über die Service-Abhängigkeiten anderer Services nicht informiert sind.

So benötigt beispielsweise ein Web Shop Service die Dienste eines Kundendaten Services. Dass dieser wiederum einen Kreditkarten Service benötigt, muss dem Entwickler des Web Shop Services nicht bekannt sein. Dem Entwickler des Kundendaten Services ist dies jedoch sehr wohl bekannt und somit ist diese Abhängigkeit im System gespeichert.

Ein ähnliches Verhalten ergibt sich, wenn die Abhängigkeiten automatisch ermittelt und absichtlich verteilt werden. Dadurch kann der Planungsaufwand im Netz verteilt werden. So können Knoten mit aktuell geringem Planungsaufwand anderen Knoten helfen.

6.5.1 Szenario verteilte Abhängigkeiten

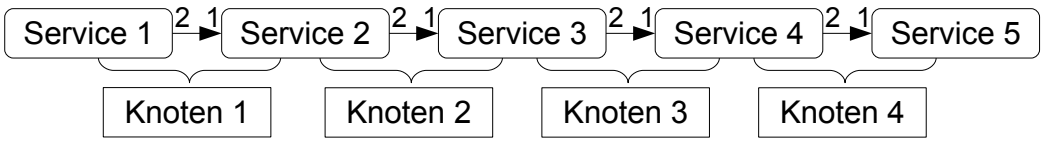


Abbildung 6.25: Abhängigkeiten auf verschiedene Knoten verteilt

Am folgenden Szenario nehmen vier Knoten teil. Der erste Anwender verwendet Knoten 1 und übergibt dem Planer Ziele bezüglich des ersten Services. Zudem ist seinem lokalen Planer bekannt, dass dieser Service von Service 2 abhängig ist. Abbildung 6.25 gibt eine Übersicht über die kompletten Abhängigkeiten, die im System vorhanden ist. So baut der Service 2 auf die Dienste des Services 3 auf. Diese Abhängigkeit ist auf dem zweiten Knoten gespeichert. Der dritte Knoten beherbergt Informationen über die Abhängigkeit von Service 3 zu Service 4. Dass Service 4 auf Service 5 basiert, ist dem Planer des vierten Knotens bekannt.

In diesem Szenario wird jeweils ein Verhältnis von zwei zu eins für die voneinander abhängigen Services angenommen. Dadurch muss bei einer Ziel-Änderung für Service 1 mehrere der anderen Services angepasst werden.

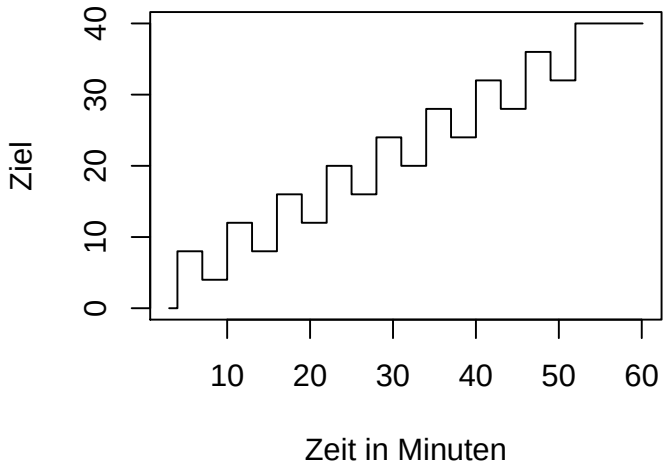


Abbildung 6.26: Verteilte Abhängigkeiten: Ziele Anwender 1

Abbildung 6.26 zeigt die Ziele des ersten Anwenders. In ihrer Form ähneln sie den Zielen des ersten Anwenders aus Abschnitt 6.3, jedoch erhöht der Anwender in diesem Szenario die Zielanzahl um acht und verringert sie dann wieder um vier. Durch das Verhältnis von 2 zu 1 folgen auch Service 2 und Service 3 dieser Form. Erst bei Service 4 und 5 ist diese Form nicht mehr beobachtbar.

Knoten zwei bis vier enthalten zwar Informationen bezüglich der weiteren Abhängigkeiten der Services, sie fügen dem System jedoch keine weiteren Ziele hinzu. In diesem Szenario fallen keine Services aus.

6.5.2 Selbst-Optimierung und Ziele der Anwender

Da es nur einen Anwender mit Zielen in diesem Szenario gibt, entspricht die durchschnittliche Auslastung des Systems in ihrer Form den Zielen dieses Anwenders.

In Abbildung 6.27 ist die Gesamtanzahl Services nach Knoten eingetragen. Wie erwartet folgt die Auslastung der einzelnen Knoten in ihrer Form den Zielen des ersten Anwenders. Wie man sieht werden die Services gleichmäßig auf alle vier Knoten verteilt. Da die Auslastung der Knoten als Summe der auf ihnen laufenden Services gemessen wird, entspricht eine Differenz von einem Service zwischen zwei Knoten immer noch einer gleichmäßige Auslastung.

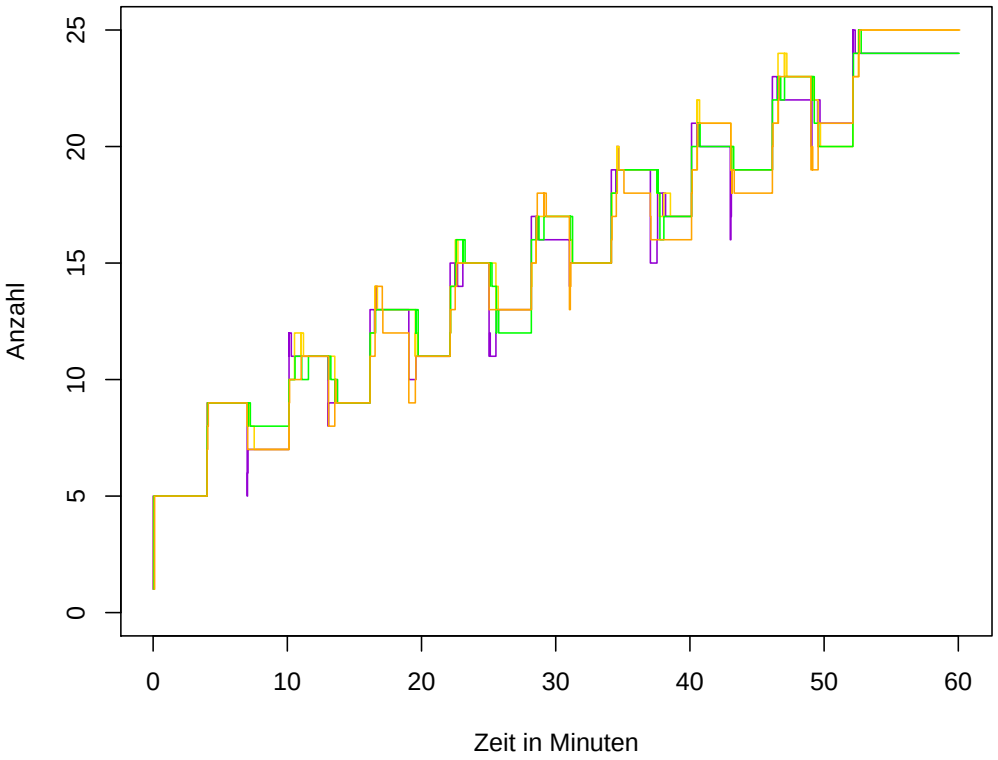


Abbildung 6.27: Verteilte Abhängigkeiten: Auslastung nach Knoten

Abbildung 6.28 zeigt die Summe an laufenden Services 1. Der Planer erfüllt die Ziele des Anwenders und verteilt die Services auf verschiedene Knoten. Die schwarze Linie symbolisiert die Ziele des ersten Anwenders. Wie man sieht werden die Ziele stets sehr schnell erfüllt, es entstehen keine zeitlichen Lücken zwischen Hinzufügen und Erfüllung des Zieles.

Die Services werden auf verschiedene Knoten verteilt. Die einzelnen Striche nach oben bzw. unten zeigen, dass ein Service von einem Knoten auf einen anderen Knoten verschoben wurde. Insgesamt laufen bis zu 40 Services 1.

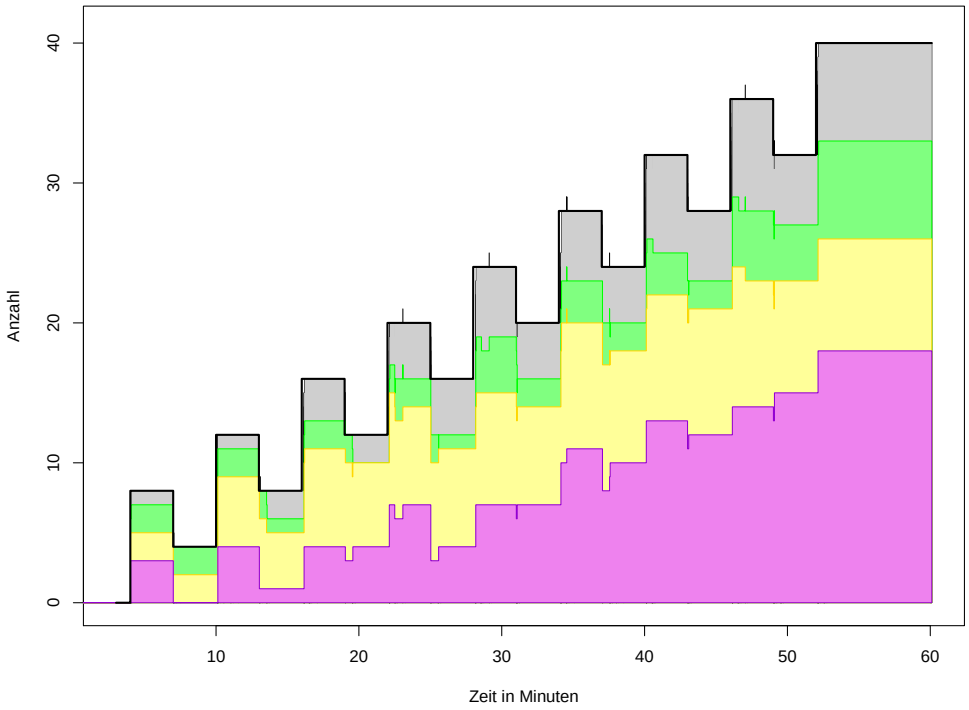
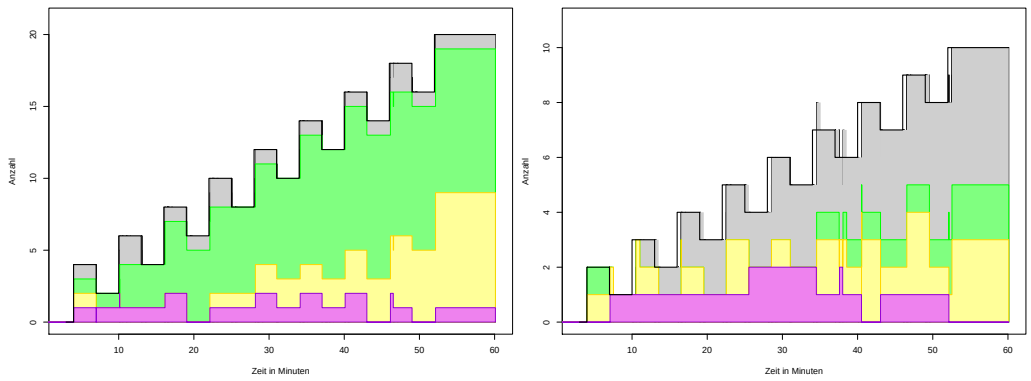


Abbildung 6.28: Verteilte Abhängigkeiten: Summe des ersten Services



(6.29.a) Summe des zweiten Services (6.29.b) Summe des dritten Services

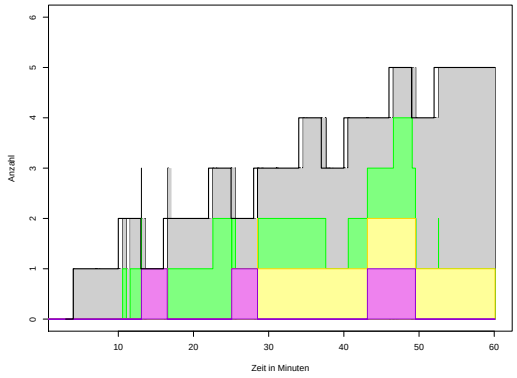
Abbildung 6.29: Verteilte Abh.: Summe des zweiten und dritten Services

Abbildung 6.29.a zeigt die Summe aller Services 2. Da ein Service 2 bis zu zwei Services 1 bedienen kann, werden bis zu 20 Services 2 gestartet. Wie auch bei Service 1 treten keine Lücke auf und der Planer erreicht die notwendige Anzahl an Services 2 schnell und zuverlässig. Dies resultiert daraus, dass sowohl Service 1 und Service 2 vom Planer des Knoten 1 gesteuert werden.

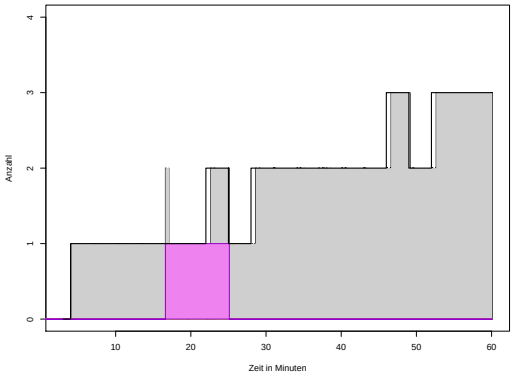
Im Gegensatz dazu, wird der dritte Service von Knoten 2 aus kontrolliert. Abbildung 6.29.b zeigt die Gesamtanzahl Services 3. Die schwarze Linie entspricht der benötigten Anzahl an Services 3, abgeleitet aus den Zielen von Service 1. Da das Hinzufügen von Zielen für Service 1 jedoch auf dem ersten Knoten geschieht, weiß der zweite Knoten dies nicht augenblicklich. Erst nachdem der erste Knoten die Services 2 gestartet hat, aktualisieren sich die Informationen auf dem zweiten Knoten und der Planer kann im nächsten Planungsschritt Service 3 starten. Dieses Verhalten führt zu Verzögerungen zwischen dem Hinzufügen und der Erfüllung der Ziele. Wie man jedoch erkennen kann, sind diese Verzögerungen auf Knoten 2 nur gering ausgeprägt.

Der Planner Manager des zweiten Knotens erreicht eine optimale Anzahl an Services 3. Zudem werden die Services auf verschiedene Knoten verteilt.

Abbildung 6.30.a zeigt den vom dritten Knoten kontrollierten Service 4. Die bereits bei Service 3 beobachtbaren Abstände zwischen dem Hinzufügen und der Erfüllung des Zieles sind auch hier zu beobachten. Zudem ändert sich die Form der schwarzen Linie im Vergleich zu den vorhergehenden Services. Der vierte Service kann jeweils zwei Services 3 bedienen. Service 3 muss zwar in einem Schritt um zwei Services erhöht, aber im nächsten Schritt lediglich um einen Service verringert werden. Dadurch entsteht die bei Service 4 beobachtbare Form.



(6.30.a) Der vierte Service



(6.30.b) Der fünfte Service

Auch Service 4 wird auf verschiedene Knoten verteilt. Die zwei Spitzen in der ersten Hälfte der Evaluierung entstehen durch Verschiebungen eines Services auf einen anderen Knoten.

Abbildung 6.30.b zeigt die Summe laufender Services 5. Er wird vom Planer des vierten Knotens kontrolliert. Die schwarze Linie entspricht der Anzahl an benötigten Services 5, abgeleitet vom ersten Service. Der vierte Knoten muss jedoch warten, bis der erste Knoten Service 1 und Service 2 gestartet hat. Anschließend startet Knoten 2 mit einer leichten Verzögerung den dritten Service. Danach passt der dritte Knoten die notwendige Anzahl an Services 4 an. Erst wenn diese Information dem vierten Knoten zur Verfügung stehen, kann der Planer die notwendigen Services 5 starten.

Reflex und Planner Manager

Durch die Verteilung der Abhängigkeiten konnte der Planungsaufwand gering gehalten werden. Obwohl die maximale Planlänge bei 20 Aktionen liegt und ein Plan durchschnittliche 3,14 Aktionen hatte, lag die maximale Erstelldauer eines Planes bei knapp 11 Sekunden. Die durchschnittliche Erstelldauer eines Planes lag bei nur 198 Millisekunden.

Der Reflex Manager benötigt in dieser Evaluierung stets weniger Zeit, als der Planner Manager. Zudem muss der Actuator nie einen Plan des Reflex Managers korrigieren. Dies ist soweit verwunderlich, da auf den Knoten zwei bis vier keine Änderungen der Ziele vorgenommen wurde. Dadurch wurde auch die Reflex Base nicht gelöscht. Somit enthielt sie Pläne, die nicht mehr aktuell waren. Dass sie nicht ausgewählt wurden, liegt an dem ihnen zugeordneten Zustand. Bei einer Ziel-Änderung auf dem ersten Knoten ergibt sich immer gleich eine Differenz von acht bzw. vier Services 1. Durch das Starten des nachfolgenden Services 2, 3, . . . ver-

größert sich diese Differenz noch. Dadurch werden die Zustände als unterschiedlich erkannt und die veralteten Pläne nicht ausgewählt.

Diese Evaluierung zeigt, dass die Verteilung von Abhängigkeiten auf verschiedene Knoten kein Problem für das System darstellt. Der Planer erfüllt die Anwenderziele und die Selbst-X Eigenschaften zuverlässig. Das Starten von nachgelagerten Services kann sich zeitlich leicht verzögern, dafür ist jedoch der Planungsaufwand pro Knoten geringer, als wenn ein Knoten für die gesamte Planung zuständig wäre.

6.6 Multiple Abhängigkeiten

In den vorhergehenden Szenarien wird angenommen, dass immer jeweils nur ein Service die Dienste eines anderen Services in Anspruch nimmt. In einem verteilten System möchten aber verschiedene Entwickler mit ihren Services die Dienste eines anderen Service in Anspruch nehmen.

So benötigt ein Web Shop beispielsweise die Dienste eines Kundendaten Services. Ein weitere Entwickler, welcher den Service einer Autovermietung programmiert, möchte diesen Kundendaten Dienst ebenfalls in Anspruch nehmen.

6.6.1 Szenario Multiple Abhängigkeiten

Abbildung 6.31 zeigt die in diesem Szenario verwendeten Abhängigkeiten. Service A1 und Service A2 sind jeweils von B1 abhängig. Ein Service B1 kann dabei zwei Services A1, zwei Services A2 oder je eine Instanz dieser beiden Services bedienen. Service B1 und Service B2 sind auf die Dienste des Services C1 angewiesen. Ein Service C1 kann dabei zwei abhängige Services versorgen.

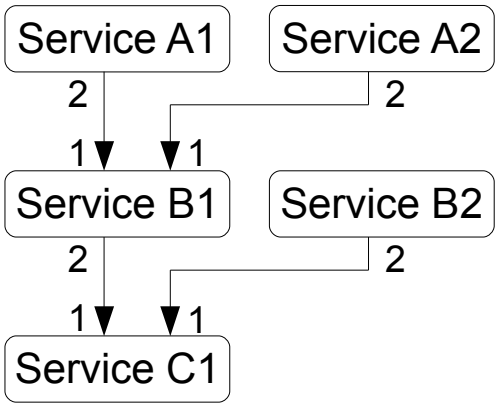
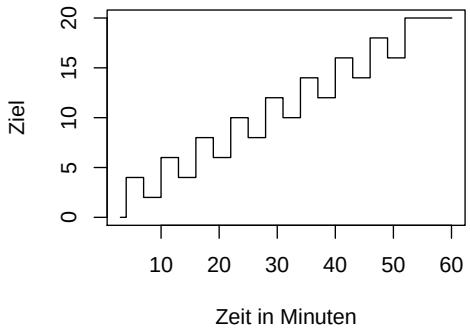
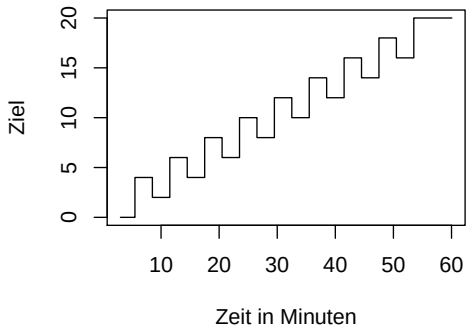


Abbildung 6.31: Multiple Abhängigkeiten

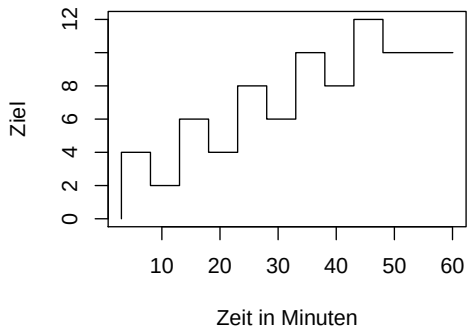
Diese Struktur verteilt sich auf drei verschiedene Knoten. Die Ziele für Service A1 werden dem ersten Knoten übergeben. Dieser Knoten ist jedoch nicht für Abhängigkeiten von Service A1 zu Service B1 zuständig. Stattdessen ist der zweite Knoten für die Abhängigkeiten A1 zu B1 und A2 zu B2 zuständig. Zudem erhält der zweite Knoten die Ziele für Service A2. Auf dem dritten Knoten werden die Ziele für den Service B2 verwaltet. Zudem sind dort die Abhängigkeiten zwischen B1 und C1 sowie zwischen B2 und C1 gespeichert. Insgesamt nehmen wieder vier Knoten an diesem Szenario teil.



(6.32.a) Ziel Service A1



(6.32.b) Ziel Service A2



(6.32.c) Ziel Service B2

Abbildung 6.32: Multiple Abh.: Ziele verschiedener Anwender

Abbildung 6.32.a zeigt die Zielanzahl des Services A1 des ersten Anwenders, während Abbildung 6.32.b die Ziele des zweiten Anwenders verdeutlicht. Beide Anwender erhöhen ihr Ziel um vier Services und verringern es im nächsten Schritt um zwei. Sie tun dies jedoch nicht gleichzeitig, sondern zeitlich versetzt. Beide Anwender starten bis zu 20 Services. Also muss der Service B1 bis zu 40 abhängige Services bedienen. Dafür werden 20 Instanzen des Services B1 benötigt.

Die Ziele des dritten Anwenders sind in Abbildung 6.32.c aufgezeichnet. Auch er erhöht die Zielanzahl um vier Services und verringert sie dann wieder um zwei. Jedoch sind die zeitlichen Abstände zwischen den Änderungen größer, als bei den beiden ersten Anwendern. Die maximale Anzahl an gewünschten Services B2 beträgt 12. Zusammen mit den bis zu 20 Instanzen des Services B1 muss der Service C1 bis zu 32 abhängige Services versorgen. Also werden bis zu 16 Service C1 benötigt.

6.6.2 Selbst-Optimierung und Ziele der Anwender

Abbildung 6.33 zeigt die Anzahl Services pro Knoten. Da alle Anwender eine tendenziell steigende Anzahl an Services wünschen, spiegelt sich dieses Verhalten auch hier wieder. Wie man sieht haben alle Knoten, inklusive des vierten Knotens, eine ähnliche Auslastung. Es sind weder Spitzen noch starke Ausreißer sichtbar. Die Selbst-Optimierung wird von allen Planern umgesetzt.

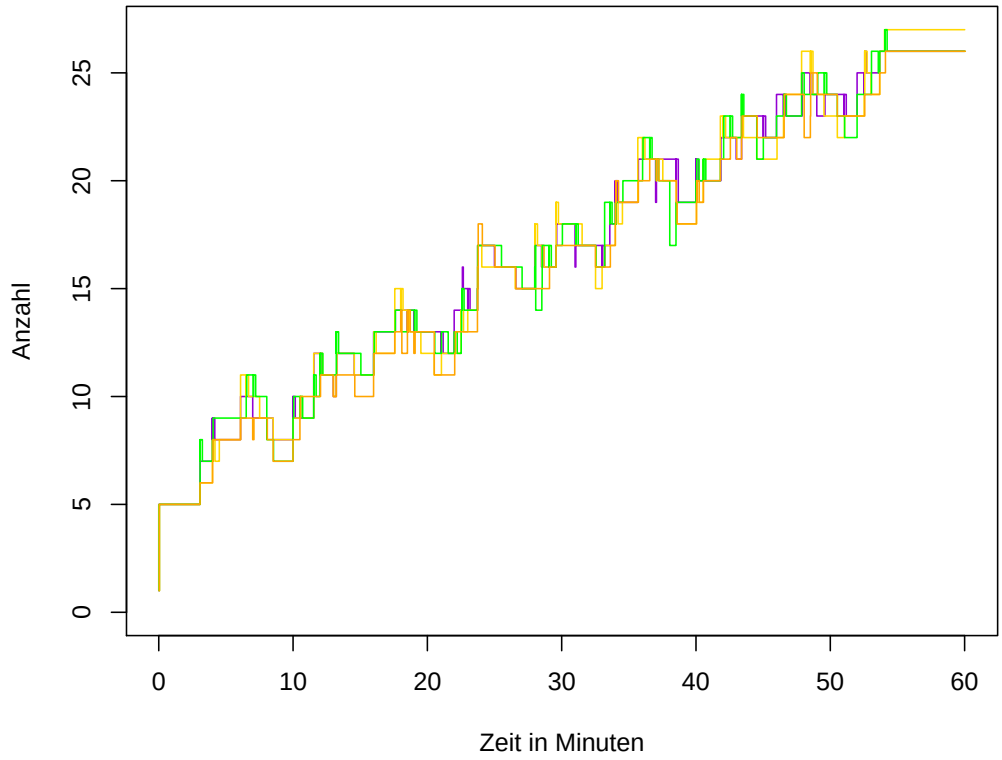


Abbildung 6.33: Multiple Abh.: Summe aller Services je Knoten

Die Ziele des ersten Anwenders sind in Abbildung 6.34 als schwarze Linie eingezeichnet. Wie man sieht, erfüllt der Organic Manager die Ziele nach dem Hinzufügen seht schnell. Es entstehen keine Lücken und es werden auch nicht zu viele Services gestartet. Der Service wird öfters verschoben, was zu den dünnen Strichen nach oben bzw. unten führt. Es wird eine Verteilung auf alle vier Knoten erreicht.

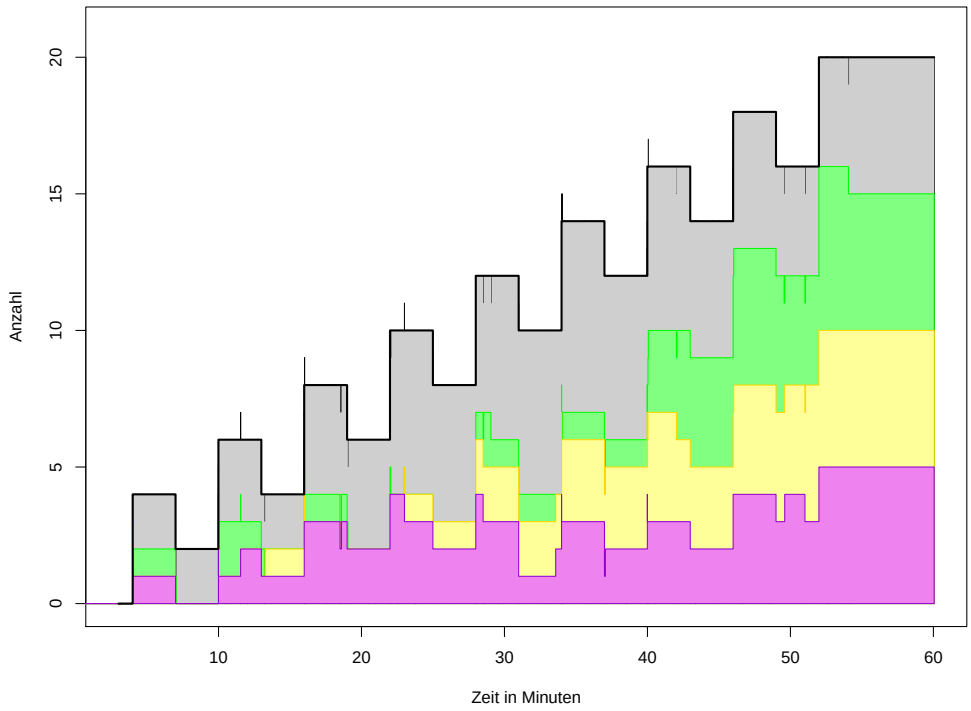
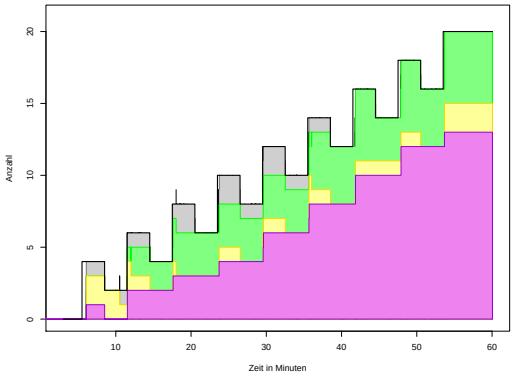
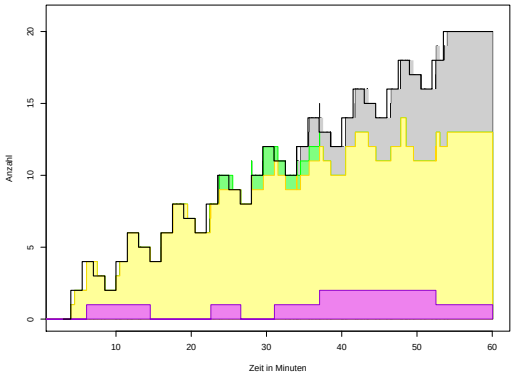


Abbildung 6.34: Multiple Abh.: Summe aller Services A1



(6.35.a) Service A2



(6.35.b) Service B1

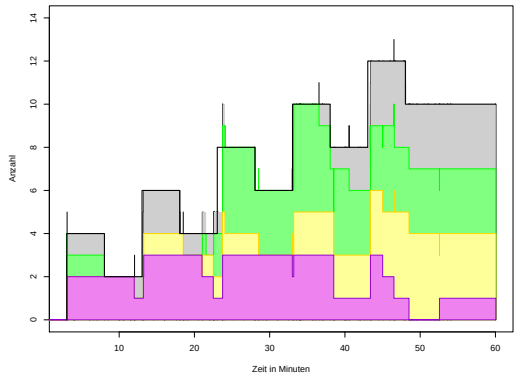
Abbildung 6.35.a zeigt die Anzahl an Services A2, welche der zweite Anwender steuert. Wie man sieht, werden die Ziele des zweiten Anwenders erfüllt. Zu Beginn der Evaluierung ist nach Hinzufügen des ersten Zieles eine kurze zeitliche Lücke zu erkennen. Hier wurde der Planer kurz vorher angestoßen und durch das Hinzufügen des Zieles die minimal erlaubte Zeitdifferenz zwischen zwei Aktivierungen des Planers unterschritten. Deshalb wurde bis zur nächsten regulären Aktivierung des Planer gewartet und das Ziel dann umgesetzt. Die Services wurden wieder auf verschiedene Knoten verteilt, wobei ein Knoten überdurchschnittlich belastet wurde. Insgesamt war seine Auslastung dadurch nicht höher, als die der anderen Knoten. Seine Auslastung wurde lediglich größtenteils durch den Service A2 verursacht.

Zusätzlich wird auf dem zweiten Knoten die Abhängigkeit von Service A1 und A2 zu Service B1 verwaltet. Abbildung 6.35.b zeigt die Anzahl an notwendigen Services B1 als schwarze Linie. Durch das Verhältnis von 2 zu 1 zu seinen abhängigen Services, folgt der Service B1 der Struktur seiner abhängigen Services. Der erste und zweite Anwender fügen ihre Ziele jedoch zeitlich versetzt ihren Planern hinzu.

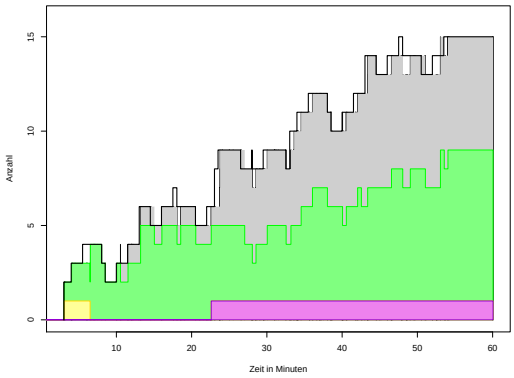
Der erste Anwender startet vier Services A1, was dazu führt, dass zwei Services B1 gestartet werden. Dann werden vom zweiten Knoten aus vier Services A2 gestartet. Entsprechend werden zwei weitere Services B1 gestartet. Als nächstes verringert der erste Anwender sein Ziel um zwei Services A1. Sobald diese Information dem zweiten Knoten vorliegt, stoppt er einen Service B1. Nach Verringerung der Zielanzahl des Services A2 durch den zweiten Anwender wird wieder ein Service gestoppt. Anschließend wieder holt sich dieser Ablauf immer wieder. So kommt die in Abbildung 6.35.b erkennbare Struktur zustande. Da der Service A1 durch den ersten Knoten kontrolliert wird, kommt es auf dem zweiten Knoten manchmal zu einer kurzen Verzögerung zwischen der notwendigen und der tatsächli-

chen Anzahl an Services B1. Sobald dem zweiten Knoten jedoch die neue Anzahl an Services A1 gemeldet wird, reagiert er entsprechend.

Wie bereits beschrieben, werden bis zu 20 Services B1 gestartet. Die Services werden wieder auf verschiedene Knoten verteilt. Als Ausgleich für die hohe Belastung mit Services A2 erhält dieser Knoten kaum Services B1. Wie bereits zu Beginn gezeigt, wird eine gleichmäßige Auslastung aller Knoten erreicht.



(6.35.c) Service B2



(6.35.d) Service C1

Abbildung 6.35.c zeigt die Gesamtanzahl an Services B2, welche durch den Organic Manager auf dem dritten Knoten kontrolliert werden. Auch hier kann der Planer die Ziele des Anwenders erfüllen. Die Services werden wieder auf die verschiedenen Knoten im Netz verteilt. Die Striche nach oben und unten zeigen, dass immer wieder Services verschoben wurden.

Der dritte Knoten ist jedoch auch für die Abhängigkeiten von Service B2 und B1 zu Service C1 zuständig. Abbildung 6.35.d zeigt die Gesamtanzahl laufender Instanzen des Services C1. Durch die Anzahl an notwendigen Services B1, kombiniert mit den Services B2 ergibt sich hier ein relativ kompliziertes Bild.

Sobald die Zielanzahl von Service B2 um vier erhöht wird, erhöht sich die notwendige Anzahl an Services C1 um zwei. Wenn sich die Zielanzahl Services B2 um zwei verringert, verringert sich die Zielanzahl Services C1 um 1. Das heißt, dass jede Änderung an der Anzahl Services B2 in diesem Fall eine Änderung der notwendigen Anzahl an Services C1 nach sich zieht.

Wenn man sich noch einmal die Zielanzahl Services B1 ansieht fällt auf, dass die Zielanzahl jeweils um zwei Services erhöht wird. Auch hier muss die notwendige Anzahl an Services C1 um 1 nach oben angepasst werden. Wenn die Anzahl an Services B1 fällt, fällt sie jedoch erst um einen Service. Hier kann die notwendige Anzahl an Services C1 noch nicht verringert werden. Erst wenn der zweite Service B1 gestoppt wird, sinkt die notwendige Anzahl an Services C1 um 1.

Zu Beginn der Evaluierung werden vier Service B2 gestartet, wodurch sich die notwendige Anzahl an Services C1 um zwei erhöht. Anschließend erhöht sich die Anzahl an Services B1 zweimal jeweils um zwei. Dadurch steigt die Anzahl an notwendige Services C1 jeweils um 1. Als nächstes verringert der dritte Anwender die Anzahl an notwendigen Services B2 und löst dadurch ein Sinken der notwendigen Anzahl an Services C1 um 1 aus. Sehr kurz darauf ist die notwendige Anzahl an Services B1 so weit gefallen, dass ein Service C1 weniger benötigt wird.

Auffällig sind die kurzen Spitzen nach oben bei Minute 17, 28 und 47. Hierbei wurde durch das Erhöhen der Anzahl an Services B1 ein erhöhter Bedarf an Services C1 ausgelöst. Sehr kurz darauf verringerte der dritte Anwender die notwendige Anzahl an Services B2 und somit sinkt der Bedarf an Services C1. In diesen Fällen war der Abstand zwischen diesen Ereignissen so kurz, dass der Planer den erhöhten Bedarf nicht berücksichtigen konnte und das System keinen zusätzlichen Service C1 gestartet hat. Da Service B1 durch den zweiten Knoten kontrolliert wird, kann es zudem zu kurzen zeitlichen Verzögerungen kommen bis dem dritten Knoten die aktuell korrekte Anzahl an laufenden Services B1 vorliegt.

6.6.3 Reflex und Planner Manager

In diesem Szenario betrug die maximale Planlänge 14 Aktionen und die durchschnittliche Planlänge 3,17 Aktionen. Die durchschnittliche Erstelldauer eines Planes ist mit 450 Millisekunden mehr als doppelt so hoch wie im vorherigen Szenario. Dadurch, dass der zweite und dritte Knoten sich jeweils um drei Services kümmern musste, war die Planungskomplexität in diesem Szenario höher als im vorhergehenden. Die Erstelldauer eines Planes war in diesem Szenario immer höher, als die vom Reflex Manager benötigte Zeit.

6.7 Fazit

Das erweiterte Modell bieten eine Vielzahl von Möglichkeiten, Services in einem Cloud-Computing-System zu steuern. Weiterhin setzt es unter anderem die Selbst-Optimierung um und erreicht eine gleichmäßige Auslastung der Knoten. Auch mit häufigen Service-Ausfällen werden die Ziele der Anwender noch erfüllt.

Die Verteilung von Abhängigkeiten ist möglich und führt zu einem geringeren Planungsaufwand für die einzelnen Knoten. Dabei kann sich die Erfüllung der Anwender-Ziele leicht verzögern.

In allen Evaluierungen benötigt der Reflex Manager weniger Zeit um einen Plan zu finden, als der Planner Manager für die Erstellung eines Planes braucht. Die Trefferrate in einigen Evaluierungen ist jedoch noch nicht optimal.

7 Erweiterungen und Optimierungen

In diesem Kapitel werden die Wartung und der Ausfall von Knoten näher betrachtet. Zudem werden die Auswirkungen der verschiedenen Parameter auf den Reflex Manager analysiert.

7.1 Knoten-Wartung & Minimierung des Energieverbrauchs

Auch in einem Organic Computing System ist es gelegentlich nötig, einzelne Knoten zu warten. Dies soll natürlich von den Anwendern nicht bemerkt werden. Deshalb dürfen auf dem betroffenen Knoten keine Anwender-Services mehr laufen. Dann kann der Administrator den Knoten warten.

Eine andere Zielsetzung in Cloud-Computing-Systemen besteht darin, Kosten zu sparen indem der Energieverbrauch minimiert wird. Dazu kann der Cloud-Computing-Anbieter ein Verfahren einführen, um einzelne Knoten abzuschalten und so Energie zu sparen. Die Knoten tauschen Informationen aus, um herauszufinden ob ein Knoten heruntergefahren werden kann, ohne die restlichen Knoten zu überlasten.

In beiden Fällen darf der ausgewählte Knoten keine Services mehr annehmen. Diese Services könnten durch das Herunterfahren des Knotens oder die Wartung Schaden nehmen.

7.1.1 Verfahren bei Knoten-Wartung

Wenn ein OCμ Knoten gewartet oder heruntergefahren werden soll, wird dies dem Organic Manager mitgeteilt. Falls der Planer des ausgewählten Knotens Ziele des Anwenders verfolgt, müssen diese Ziele an andere Knoten übertragen werden. Anschließend wird das aktive PDDL Modell zur Laufzeit ausgetauscht. Zudem werden eingehende Service Verschiebungen abgelehnt. Der betroffene Knoten informiert die restlichen Knoten. Sie schließen den Knoten anschließend aus ihren zukünftigen Planungen aus. Mit Hilfe des neuen PDDL Modells werden auf dem Knoten laufende Anwender-Services auf andere Knoten verschoben.

7.1.2 Evaluierung

Am folgenden Szenario nehmen zehn Knoten und 60 Anwender-Services teil. Im Laufe der Zeit soll Knoten 1, 2 und 3 gewartet werden.

Die 60 aktiven Services werden von keinem Planer kontrolliert. So kann gezeigt werden, dass auch ohne eine explizite Überwachung kein Service verschwindet. Zu Beginn starten Knoten 1, 2 und 3 je 20 Services. Die Planner Manager aller Knoten haben dasselbe Modell und versuchen die Last gleichmäßig zu verteilen. Abbildung 7.1 zeigt die Auslastung der einzelnen Knoten. Zu Beginn steigt die

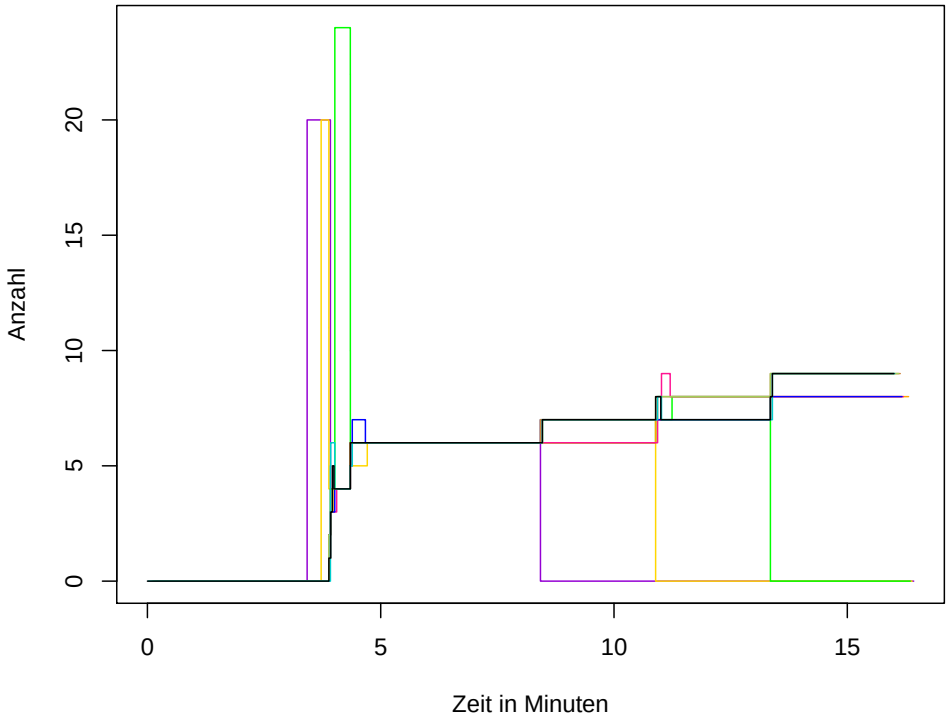


Abbildung 7.1: Wartung: Summe Anwender-Services je Knoten

Auslastung der ersten drei Knoten durch das Starten von je 20 Services deutlich. Bei Minute fünf haben die Planer die 60 Services jedoch gerecht zu je sechs Services auf die zehn Knoten verteilt. In der achten Minute erhält der erste Knoten den Befehl zur Wartung. Er tauscht sein PDDL Modell aus und verschiebt seine Services auf die anderen neun Knoten im Netz. Dabei werden die Knoten gleichmäßig belastet.

In der 11. Minute erhält Knoten zwei und in der 13. Minute Knoten drei den Befehl zur Wartung. Auch sie verteilen ihre Services auf die übrigen Knoten. Am

Ende verteilen sich die 60 Services auf sieben Knoten und führen zu einer Auslastung von 7 bzw. 8 Services je Knoten.

Das System kann die Selbst-Optimierung umsetzen und die betroffenen Knoten komplett aus ihrer Betrachtung ausschließen. Zwar führt die Wartung einzelner Knoten zu einer erhöhten Last für die restlichen Knoten, eine gleichmäßige Auslastung erreicht das System dennoch.

Die Summe aller laufenden Instanzen des Anwender-Services ist in Abbildung 7.2 dargestellt. Die Farbe entspricht den Knoten, auf dem die Services laufen. Wie man sieht, sind ständig 60 Services aktiv. Die untersten drei Farbschichten gehören

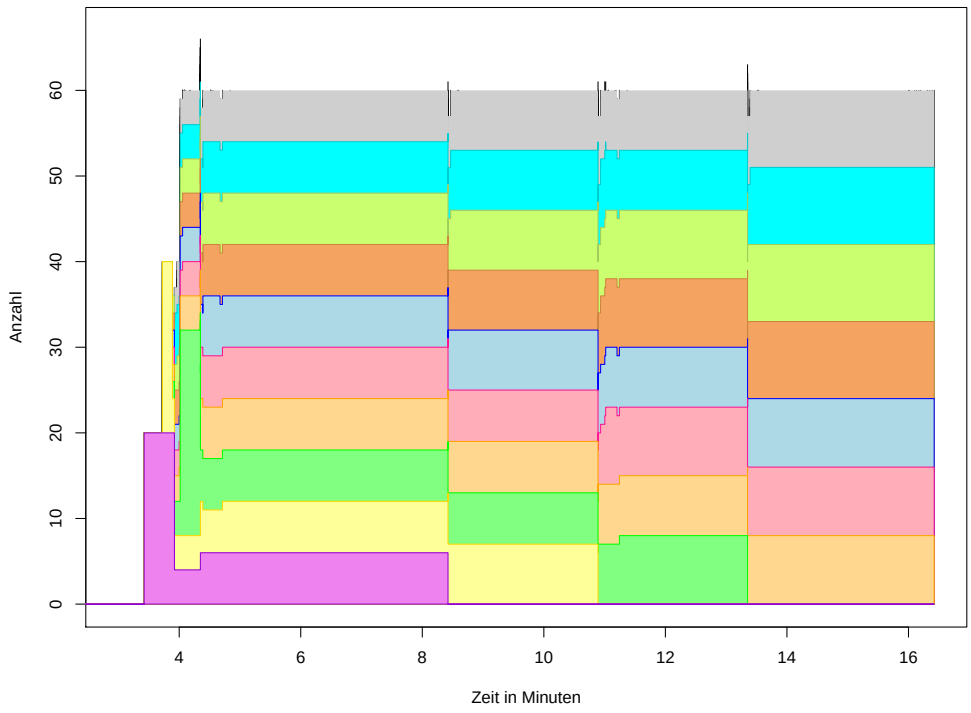


Abbildung 7.2: Wartung: Summe aller Anwender-Services

zu den Knoten $1 - 3$. Nach Erhalt des Wartung-Befehls verschieben sie jeweils ihre Services und verschwinden aus dieser Abbildung. Die kurzen Ausschläge nach oben und unten werden von Verschiebungen von Services ausgelöst. Bei jeder Wartung erhalten die restlichen Knoten mehr Last. Während dieses Vorgangs bleiben alle 60 Services erhalten. Kein Service fällt aus oder muss neu gestartet werden.

Die Trefferrate der Reflex Manager der einzelnen Knoten ist in Abbildung 7.3 aufgetragen. Die hohe Rate von 60% entspricht ca. 20 Treffer. Da Knoten $1 - 3$ kürzer am Netz teilnehmen ist ihre Trefferrate niedriger.

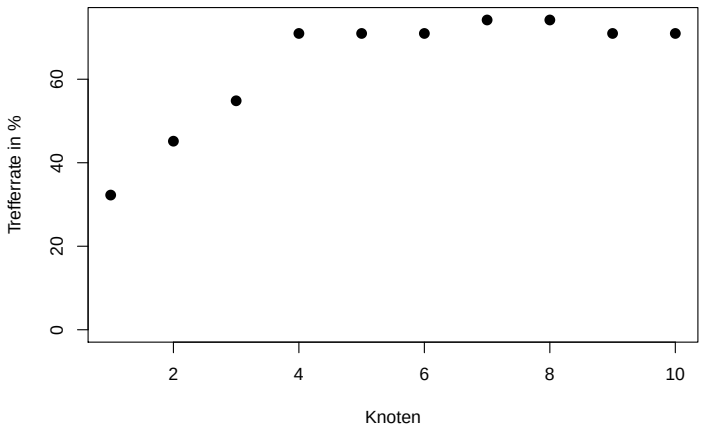


Abbildung 7.3: Wartung: Trefferrate Reflex Manager

In diesem Szenario wurden die Organic Manager über eine bevorstehende Wartung informiert. Die Anwender-Services können verschoben werden und der Knoten kann kontrolliert heruntergefahren werden. Computersysteme können jedoch auch abstürzen. Durch einen Hardware- oder Software-Fehler sind im schlimmsten Fall alle Daten auf diesem System verloren. In einem System aus OCμ Knoten können die negativen Folgen eines unerwarteten Ausfalls jedoch verringert werden. Im nächsten Abschnitt wird der Ausfall von Knoten simuliert.

7.2 Knotenausfall

Bei Ausfall eines Knotens sind alle auf ihm aktiven Anwender-Services betroffen. Zudem sind die Ziele des lokalen Planers verloren. Von Trumler et al. [44] wurde für OCμ ein verteilter Service zur Datenspeicherung entwickelt. Die dort abgelegten Daten werden automatisch auf mehreren Knoten gespeichert. Die Datenspeicher synchronisieren sich selbständig.

Die Ziele der Anwender können in diesem Datenspeicher abgelegt werden. Zudem kann abgespeichert werden, welcher Knoten für welches Ziel zuständig ist. Fällt ein Knoten aus, kann mit Hilfe des Datenspeichers herausgefunden werden, welche Ziele betroffen sind. Diese können dann an einen oder mehrere andere Knoten verteilt und aktiviert werden.

Die Anwender-Services auf dem betroffenen Knoten fallen jedoch aus. Sie werden aber im Regelfall von einem Organic Manager überwacht. Er erkennt den Ausfall und startet sie neu.

7.2.1 Szenario Knotenausfall

Das folgende Szenario basiert auf dem Cloud-Computing-Szenario aus Abschnitt 6.3, Seite 62. Es nehmen drei Anwender und zehn Knoten an diesem Szenario teil und die Services sind voneinander abhängig. Die Anwender ändern im Laufe der Zeit ihre Ziele. Die Ziele der drei Anwender werden den Planern auf den ersten drei Knoten übergeben.

Der erste Anwender steigert tendenziell seinen Bedarf an Services, wobei er ihn zwischendurch auch wieder senkt. Der zweite Anwender erhöht die Anzahl an notwendigen Services regelmäßig, um anschließend eine periodisch schwankende Anzahl an Services zu fordern. Der dritte Anwender steigert die gewünschte Anzahl an Services schnell, um anschließend eine tendenziell fallende Anzahl an Services festzulegen.

Der einzige Unterschied zum originalen Szenario ist, dass diesmal die Knoten 7-10 im Laufe der Zeit ausfallen. Die anderen Knoten bemerken diesen Ausfall und kompensieren ihn. In diesem Szenario wird der Ausfall eines Knoten simuliert, indem er keine eingehende Service-Verschiebungen mehr akzeptiert und alle lokal laufenden Anwender-Services beendet.

7.2.2 Evaluierung

Abbildung 7.4 zeigt die Anzahl laufender Anwender-Services je Knoten. Bei Minute 20 fällt der achte Knoten aus. Der neunte Knoten fällt in Minute 30 und der zehnte in Minute 40 aus. Alle anderen Knoten laufen normal weiter. Die Last teilt sich gleichmäßig unter ihnen auf. Durch die Ausfälle ist die Last der übrigen Knoten am Ende der Evaluierung um drei Services höher, als sie es ohne die Ausfälle wäre.

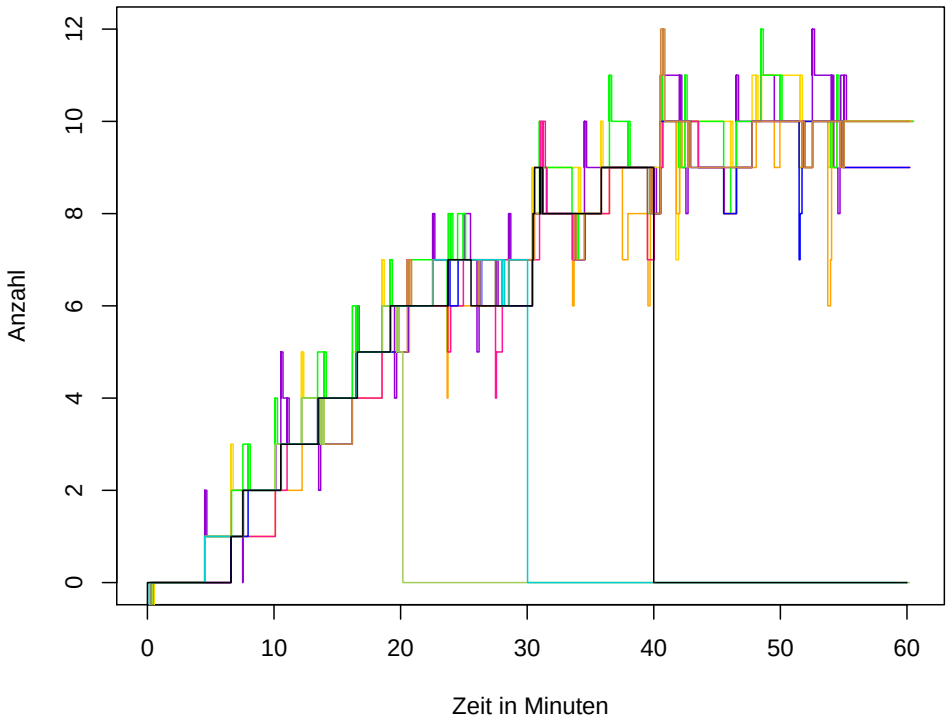


Abbildung 7.4: Knotenausfall: Summe aller Anwender-Services je Knoten

Die einzelnen kurzen Ausschläge nach unten werden von den Anwendern verursacht. Sie verringern die Zielanzahl eines Services. Der Planer stoppt anschließend einige Services. Da ein bestimmter Service nicht auf allen Knoten gleich oft läuft, verringert sich die Last auf manchen Knoten stärker als auf anderen. Die restlichen Knoten bemerken dies und verschieben Services auf den betreffenden Knoten.

Der Service A3 ist am stärksten von den Knotenausfällen betroffen. In Minute 20 laufen zwei Instanzen des Services A3 auf dem achten Knoten. Bei Ausfall des neunten und zehnten Knoten sind je drei Instanzen des Services A3 betroffen. Abbildung 7.5 zeigt die Anzahl an laufenden Services A3.

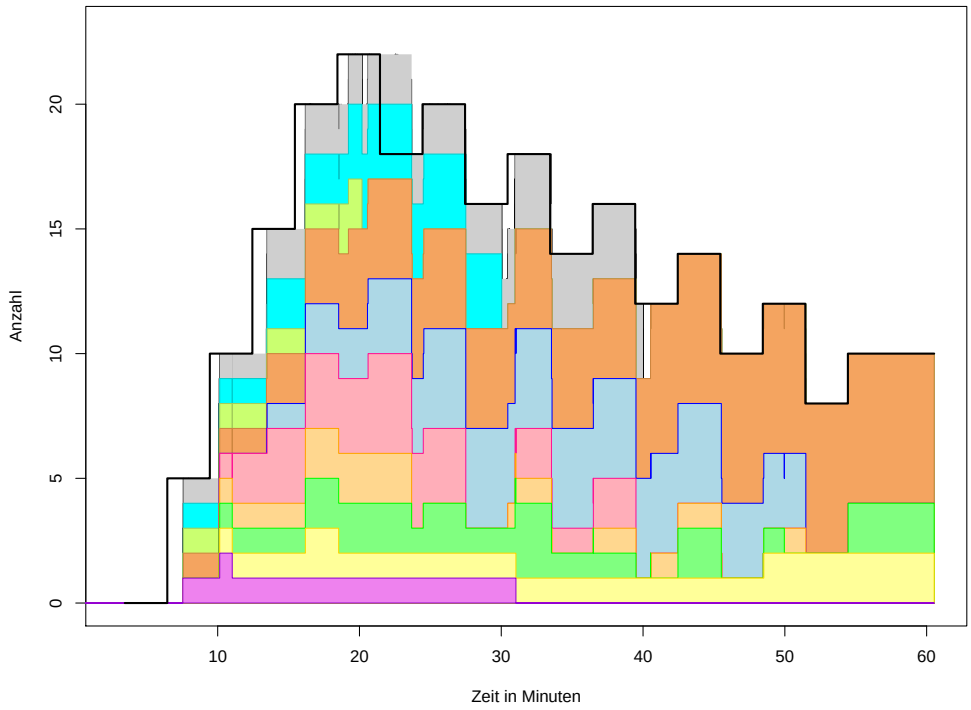


Abbildung 7.5: Knotenausfall: Summe aller Services A3

Nach Hinzufügen eines Zieles zeigt sich ein kurzer zeitlicher Abstand bis die nötigen Services tatsächlich gestartet sind. Dies liegt daran, dass der Planer bereits kurz vor dem Hinzufügen angestoßen wurde und mit dem Planen noch kurz zu wartet, damit der eben erstellte Plan erst ausgeführt werden kann.

Bei Minute 40 fällt der zehnte Knoten aus und die oberste Farbschicht verschwindet. Es entsteht eine kurze Lücke bis der Ausfall erkannt und die Services neu gestartet werden. Bei Minute 30 trifft es den neunten Knoten und damit die zweitoberste Farbschicht. Der achte Knoten wird von der drittobersten Farbschicht dargestellt. Nach den jeweiligen Ausfällen sieht man eine kurze Zeitspanne, in der die ausgefallenen Services fehlen. Dies wird jedoch schnell erkannt und die Services neu gestartet.

Alle anderen Services sind weniger stark betroffen. Ausgefallene Services werden erkannt und neu gestartet. Die Services B1, B2, B3, C1 und C2 folgen den Zielen der Anwender und zeigen keine Auffälligkeiten.

Der Ausfall einzelner Knoten hat als hauptsächliche Folge den Ausfall aller darauf laufenden Services. Dieser Ausfall wird zeitnah erkannt und die Services neu gestartet. Die übrigen Knoten müssen die erhöhte Last auffangen.

7.3 Optimierung Reflex Manager

Der Reflex Manager speichert Pläne des Planers und verwendet den Systemzustand als Schlüssel. Tritt ein ähnlicher Zustand auf, kann ein bereits gespeicherter Plan wiederverwendet werden. Durch den Reflex verbessert sich die Reaktionszeit des Systems. Der Reflex Manager verfügt über mehrere Variablen, die Einfluss auf seine Trefferrate haben:

- Anzahl gespeicherter Pläne
- Verdrängungsstrategie
- Metrik zum Vergleich von Zuständen
- Schwellwert, ab dem Zustände als ähnlich gelten

Der Reflex Manager speichert nur eine gewisse Anzahl an Plänen. Werden wenig Pläne gespeichert ist die Trefferrate gering. Eine große Reflex Base benötigt jedoch Speicherplatz und der Zugriff verlangsamt sich.

Wenn die maximale Anzahl an Pläne erreicht ist, gibt eine Verdrängungsstrategie das weitere Vorgehen vor. Sie entscheidet, ob ein neuer Zustand gespeichert wird und welcher alte dafür gelöscht wird. Die implementierten Verdrängungsstrategien sind in Abschnitt 4.2.4 ab Seite 41 beschrieben.

Um zwei Zustände miteinander zu vergleichen, wird die Metrik 1 aus Definition 2 von Abschnitt 4.2.3, Seite 40 verwendet. Ist die Differenz der Zustände niedriger als ein gewisser Schwellwert, gelten die Zustände als ähnlich.

7.3.1 Evaluierung Reflex Manager

In den Evaluierungen aus Kapitel 5 wird der Planner Manager alle 30 Sekunden angestoßen und erzeugt somit nur eine begrenzte Menge an Fällen, an denen der Reflex Manager gemessen werden kann.

Um die Anzahl an Fällen zu erhöhen und mehrere Verdrängungsstrategien und Reflex-Base-Größen gleichzeitig testen zu können, werden die Systemzustände simuliert. Der Planner Manager erstellt basierend auf diesem Zustand einen Plan. Der generierte Zustand wird anschließend verschiedenen Reflex Managern übergeben. Falls möglich, liefern sie einen Plan. Beide Pläne werden verglichen und das Ergebnis gespeichert. Da in allen Evaluierungen aus Kapitel 5 der Reflex Manager deutlich schneller war als der Planer, wird bei einem Vergleich davon ausgegangen, dass der Plan des Reflex Managers bereits komplett ausgeführt wurde.

In dieser Evaluierung liegt dem Planer das erweiterte Modell aus Kapitel 6 zu Grunde. Es werden drei verschiedene Services verwendet. Service A1 ist von Ser-

vice B1 abhängig. Ein Service B1 kann bis zu zwei Instanzen des Services A1 bedienen. Dem Planer wird vorgegeben, dass 20 Services A1 laufen sollen.

Wenn die Systemzustände komplett zufällig generiert werden, zeigt sich die Verdrängungsstrategie *Random* als sehr erfolgreich. Da ein komplett zufälliges Verhalten für ein echtes System jedoch unwahrscheinlich ist, werden die Systemzustände nach gewissen Regeln generiert.

Für die Simulierung der Systemzustände dient ein Standardzustand als Ausgangspunkt. In einem echten System wählt der Planer in jeder Runde Aktionen aus, um einen Zielzustand zu erreichen. Meistens benötigt der Planer nur eine Runde um das System in diesen Zielzustand zu versetzen. Anschließend verändert sich das System und der Planer versucht erneut einen Plan zum Zielzustand zu finden. Die Veränderungen in einem System nehmen jedoch selten extreme Ausmaße an. Deshalb kann man davon ausgehen, dass öfters der gleiche Zustand auftritt. Um diesem Umstand Rechnung zu tragen, werden in 10% aller Fälle der Standardzustand ohne Modifikationen verwendet.

Tabelle 7.1 zeigt die Werte des Standardzustandes und mögliche Schwankungen. Die Anzahl an Knoten beträgt immer 10, auf dem simulierten Knoten laufen keine Services A1 und B1. Wenn der Standardzustand manipuliert wird, wird die Gesamtanzahl an Services A1, B1 oder die lokale Anzahl an Services A2 verändert. Zusätzlich wird die Summe der Gesamtanzahl Services A1, B1 und A2 berechnet und zufällig um bis zu drei Services verändert. Die Gesamtanzahl Services A2 und die lokale Anzahl Services A2 werden je nach Szenario im Zeitverlauf verändert, um bestimmte Verhalten nachzubilden.

	Standardzustand	Schwankungsbreite
Zufällige Werte		
Gesamtanzahl Services A1	20	18-20
Gesamtanzahl Services B1	10	8-12
Im Zeitverlauf ändernde Werte		
Gesamtanzahl Services A2 zu Beginn	50	
Lokale Anzahl Services A2 zu Beginn	8	6-12
Anzahl Services im Netz zu Beginn	80	77-83
Fixe Werte		
Anzahl Knoten	10	
Lokale Anzahl Services A1	0	
Lokale Anzahl Services B1	0	

Tabelle 7.1: Standardzustand und mögliche Schwankungen

Nach Generierung eines Zustandes wird er dem Planer übergeben. Zudem werden mehrere Reflex Manager mit unterschiedlichen Größen und verschiedenen Verdrängungsstrategien mit dem Zustand konfrontiert. So kann sichergestellt werden, dass trotz der zufälligen Generierung die Ergebnisse der verschiedenen Reflex Manager vergleichbar sind. Anschließend werden die Ergebnisse der Reflex Manager mit dem Plan des Planers verglichen.

7.3.2 Steigende Last

In diesem Szenario wird eine ständig steigende Auslastung des Systems angenommen. Alle 50 Schritte wird die Anzahl an laufenden Services A2 des Standardzustands um zehn und die lokal laufenden Services A2 um 1 erhöht. Insgesamt dauert eine Evaluierung 1000 Schritte.

Auf der Y-Achse ist die Trefferrate angetragen. Wenn der Reflex Manager einen Plan findet, kann dieser mit dem originalen Plan absolut übereinstimmen. Dies ist in der Abbildung in Gelb als *Korrekt* markiert. Kann ein *Wechsel* auf den Plan des Planers erfolgen, ist dies in Orange eingezeichnet. Rot bedeutet, der Reflex Manager hat einen falschen Plan zurückgeliefert.

Abbildung 7.6 zeigt die Trefferrate eines Reflex Managers, der keine Pläne löscht, da sein Speicher nicht begrenzt ist. Die Zahlen an der X-Achse entsprechen den Schwellwerten der Metrik, ab dem zwei Zustände als ähnlich erkannt werden. Da hier keine Pläne gelöscht werden, stellt dies die Obergrenze der möglichen Treffer dar. Es wird eine Trefferrate von bis zu 80% bei einer Fehlerquote von ca. 10% erreicht.

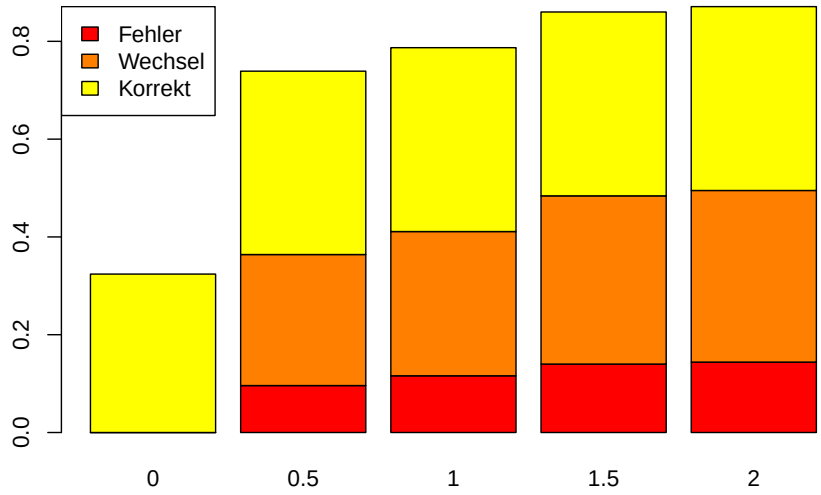


Abbildung 7.6: Trefferrate bei steigender Last ohne Löschung

Mit der euklidischen Metrik aus Abschnitt 4.2.3 und einem Schwellwert von 1, ergibt sich für einen Reflex Manager der Größe 50 das in Abbildung 7.7 dargestellte Ergebnis.

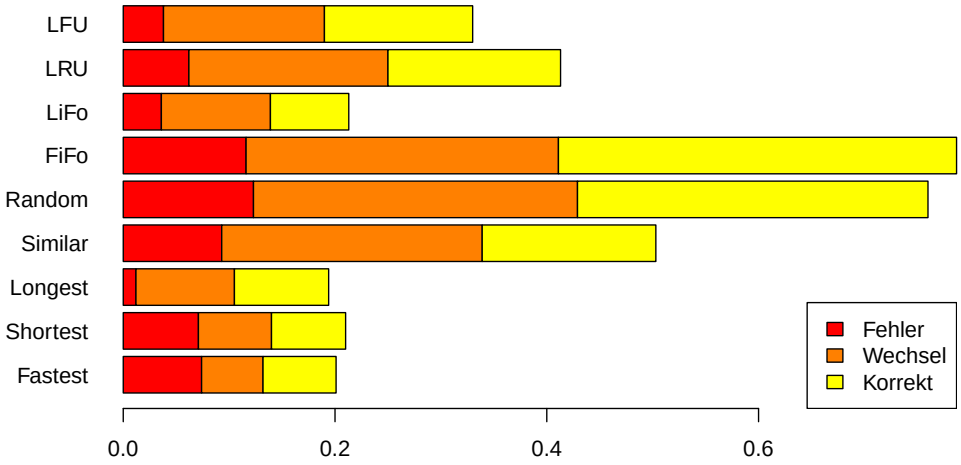


Abbildung 7.7: Trefferrate bei steigender Last

Durch die Erhöhung der Last alle 50 Schritte können alte Pläne nicht mehr wiederverwendet werden. *FiFo* liefert somit die höchste Trefferrate, gefolgt von *Random*. Bei allen Strategien kann ein großer Teil der Pläne durch einen Wechsel auf den Plan des Planers ergänzt werden. Im Folgenden werden die beiden besten Strategien genauer betrachtet.

Abbildung 7.8 zeigt das Verhalten der Verdrängungsstrategie *Random*. Auf der unteren X-Achse sind verschiedene Schwellwerte angetragen und auf der Y-Achse die damit erzielte Trefferrate. Für jeden Schwellwert gibt es vier verschiedene Reflex Manager mit maximal 10, 30, 50 und 100 gespeicherten Plänen, wie an der oberen X-Achse ersichtlich ist.

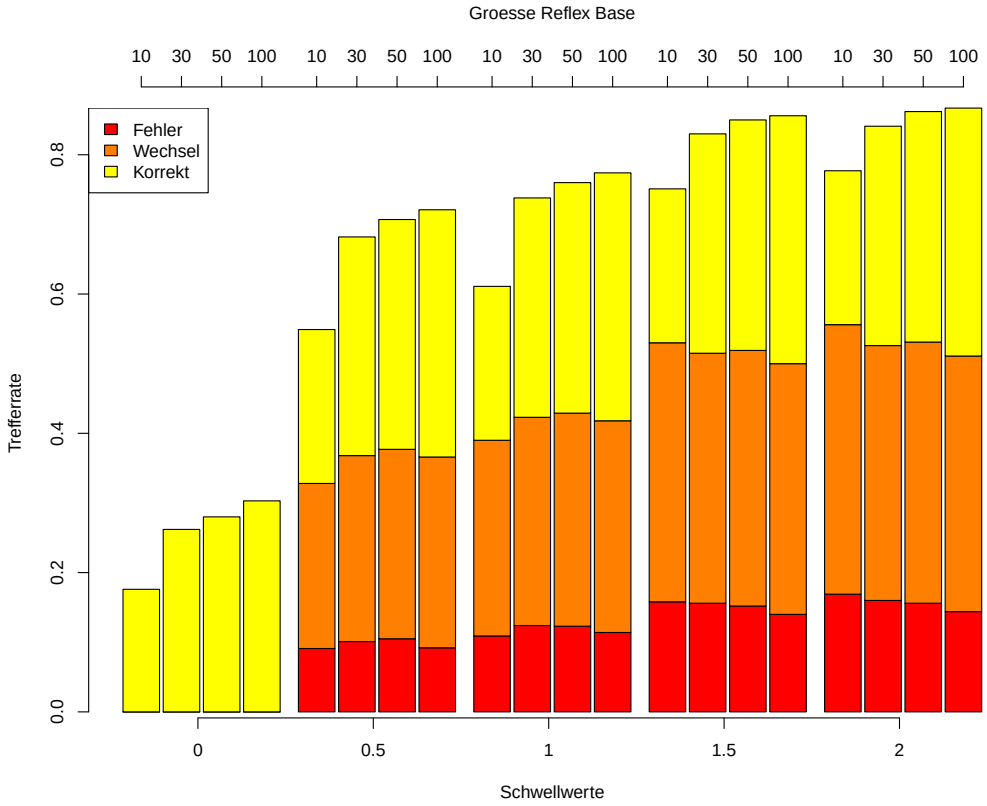


Abbildung 7.8: Trefferrate bei steigender Last: Random

Wie bei allen anderen Strategien steigt die Trefferrate, je mehr Pläne gespeichert werden können. Sie steigt jedoch nicht linear. Stattdessen bringt eine Verdoppelung der Größe von 50 auf 100 Pläne nur eine geringe Verbesserung. Die Trefferrate der kleineren Reflex Bases sind bereits relativ gut. Eine Vergrößerung der Reflex Base kann also gar keine lineare Steigerung der Trefferrate mehr bewirken.

Die stärkste Veränderung der Trefferrate findet man zwischen einem Schwellwert von 0 und 0,5. Abbildung 7.9 zeigt diesen Bereich für die Strategie *FiFo* etwas genauer. Eine Besonderheit dieser Strategie ist, dass sie im vorliegenden Szenario bereits bei 30 Plänen ein fast ebenso gutes Ergebnis wie mit 100 Plänen liefert. Bei einem Schwellwert von 0 werden nur exakt gleiche Zustände als ähnlich erkannt und somit nur eine geringe Trefferrate erzielt. Mit steigendem Schwellwert erhöhen sich auch die Trefferraten. Jedoch treten auch mehr Fehler auf.

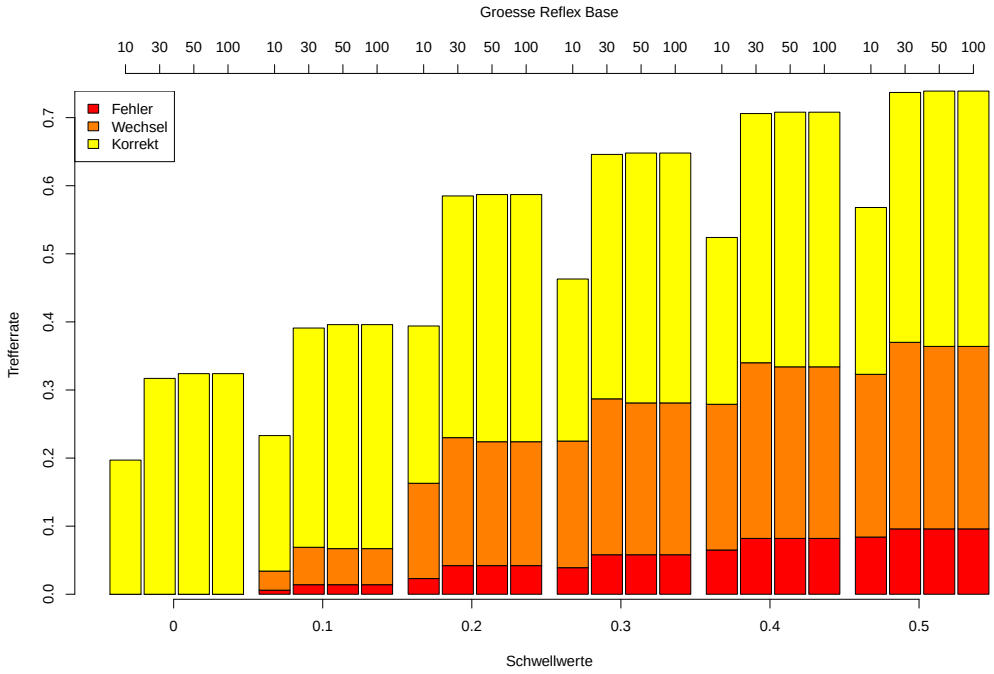


Abbildung 7.9: Trefferrate bei steigender Last FiFo

In einem Szenario in dem sich alte Zustände nicht wiederholen sind *FiFo* oder *Random* eine gute Wahl. Beide einfach zu implementierende Verdrängungsstrategien liefern in einem solchen Fall gute Trefferraten.

7.3.3 Zyklisches Verhalten

In vielen Situationen kann ein zyklisches Verhalten beobachtet werden. In solchen Fällen treten bereits beobachtete Zustände später wieder auf.

Im folgenden Szenario wird die Anzahl laufender Service alle 50 Schritte um je zehn Services verändert. Dreimal hintereinander wird die Last erhöht und dann dreimal verringert. Dieser Zyklus wiederholt sich mehrmals.

Die maximal erreichbare Trefferquote steigt, da sich Zustände wiederholen können. Abbildung 7.10 zeigt einen Reflex Manager, der keine Pläne lösche. Bereits ein Schwellwert von 0 ergibt eine Trefferquote von 70%. Dieses Ergebnis ist mit Reflex Managern, denen nur eine begrenzte Kapazität an speicherbaren Plänen zur Verfügung steht, nicht zu erreichen.

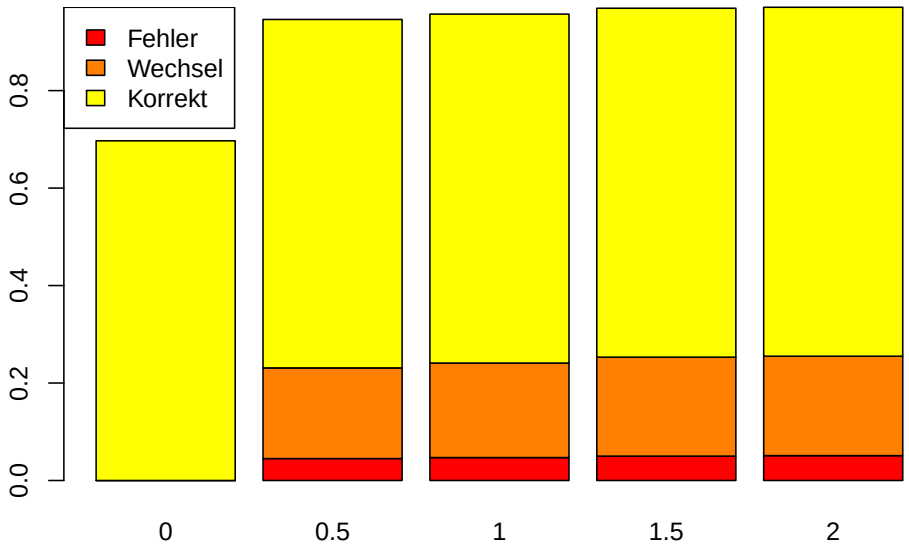


Abbildung 7.10: Trefferrate bei steigender Last ohne Löschung

Abbildung 7.11 zeigt die Trefferraten der einzelnen Strategien bei maximal 50 gespeicherten Plänen pro Reflex Manager. Zudem wurde die euklidische Metrik zum Vergleich von Zuständen und ein Schwellwert von 1 verwendet. In diesem Fall er-

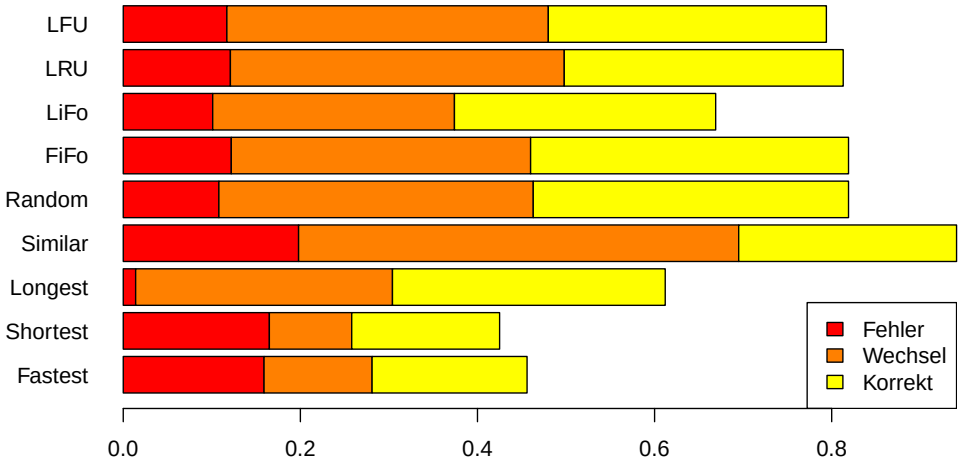


Abbildung 7.11: Trefferrate bei zyklischer Last

zielt *Similar* die beste Trefferrate. Bei dieser Strategie wird eine hohe Abdeckung der möglichen Zustände erzielt, indem ähnliche Zustände verdrängt werden. Dafür ist die Anzahl an Treffern, bei denen ein Wechsel möglich ist besonders hoch.

Auffallend ist die geringe Fehlerrate der Strategie *Longest*, jedoch hat sie eine geringe Trefferrate. Diese Strategie bevorzugt die Löschung von langen Plänen. Gespeichert werden eher kurze Pläne mit wenigen Aktionen. Dadurch kann der vom Reflex Manager vorgeschlagene Plan leichter ergänzt werden und es entsteht eine geringere Fehlerrate.

Bei *Shortest* ist genau das Gegenteil der Fall. Da die gespeicherten Pläne eher länger sind, steigt das Risiko für einen Fehler. Es kann nicht mehr auf den korrekten Plan gewechselt werden.

Die größten Veränderungen der Trefferrate treten zwischen einem Schwellwert von 0 und 0,5 auf. Abbildung 7.12 zeigt die Trefferrate der Verdrängungsstrategie *Similar* für verschiedene Reflex-Base-Größen. Die Trefferrate steigt deutlich mit der maximalen Größe der Reflex Base. Leider steigt auch die Fehlerrate.

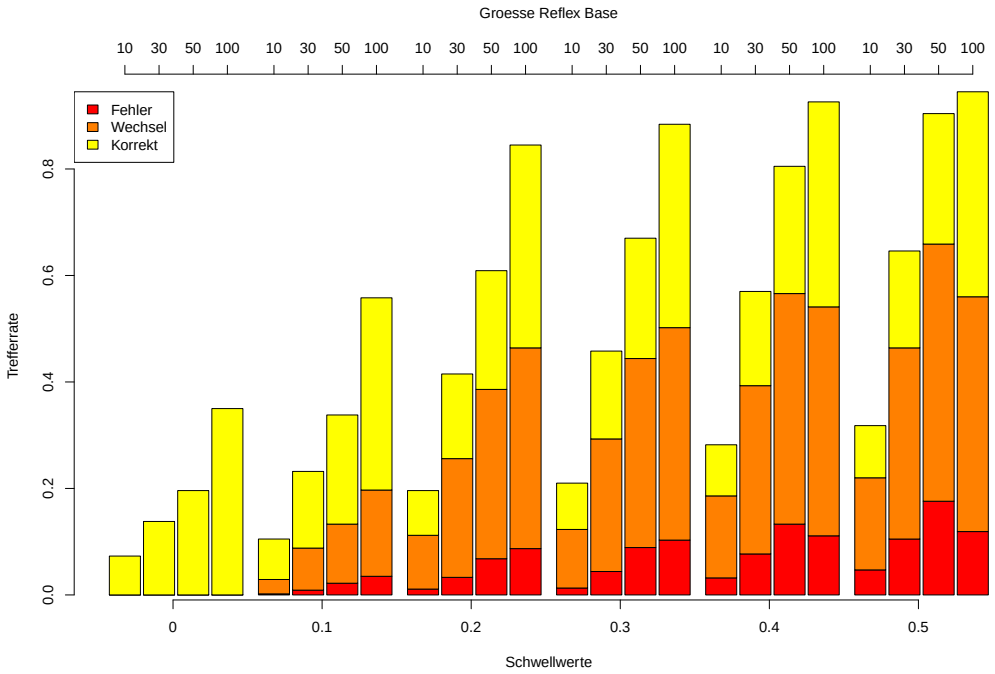


Abbildung 7.12: Trefferrate bei zyklischer Last: Similar

Die Verdrängungsstrategie *Longest* zeigt eine geringe Fehlerrate. Abbildung 7.13 zeigt ihr Verhalten unter verschiedenen Schwellwerten und Größen. Im Gegensatz zu den anderen Strategien steigt die Trefferrate mit höheren Schwellwerten nur langsam an. Die Fehlerrate bleibt jedoch gering. Bei einer Reflex Base Größe von 50 und einem Schwellwert von 2 erzielt diese Strategie eine sehr gute Trefferrate.

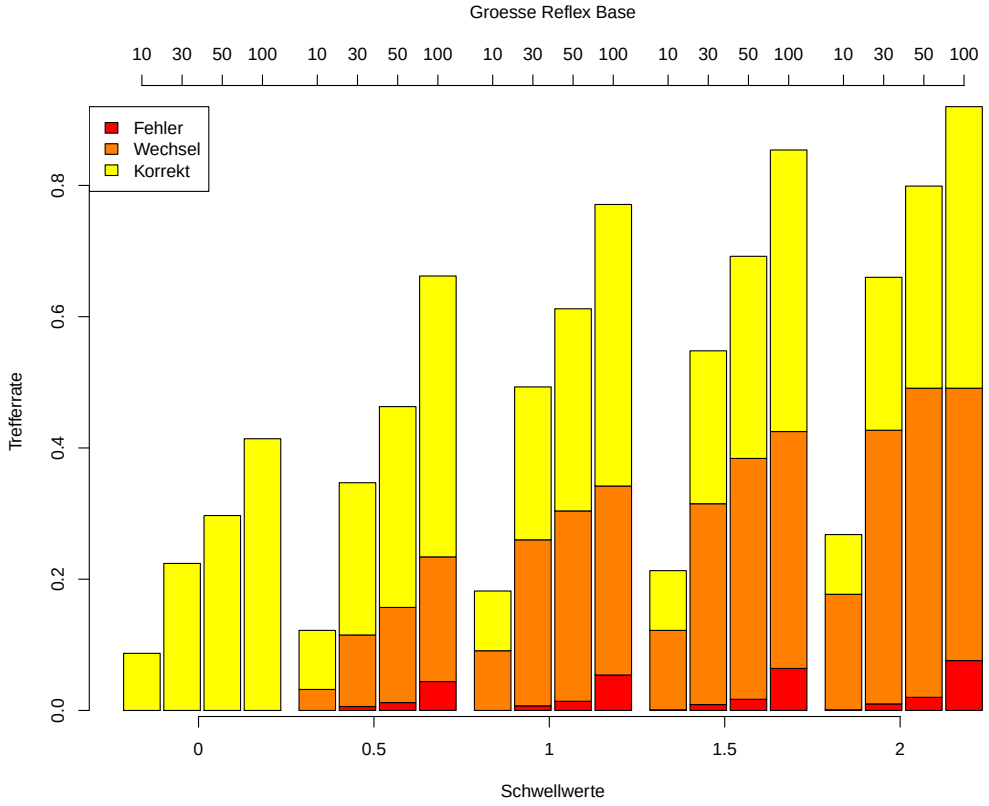


Abbildung 7.13: Trefferrate bei zyklischer Last: Longest

7.3.4 Fazit

Die verschiedenen Einflussfaktoren auf die Trefferrate des Reflex Managers wurden mit Hilfe einer Simulation evaluiert. Wie sich gezeigt hat, ist bei einem System mit komplett unbekanntem Verhalten eine zufällige Verdrängungsstrategie eine gute Ausgangsbasis. Wenn sich das System ändert, ohne alte Zustände zu wiederholen, eignet sich die FiFo Strategie. Die Verdrängungsstrategie Similar liefert eine gute Trefferrate, wenn ein zyklisches System-Verhalten beobachtbar ist. Sie löscht Pläne für ähnliche Zustände und sorgt so für eine gleichmäßigere Abdeckung der möglichen Zustände.

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

Die Komplexität und damit der Verwaltungsaufwand moderner Systeme steigt ständig. Neue Software- und Hardware-Entwicklungen tragen ebenso wie die steigende Anzahl an Komponenten dazu bei. OCμ ist eine Middleware für verteilte Systeme, die durch Selbst-Organisation den Administrationsaufwand senkt. In dieser Arbeit wird als Anwendung ein Cloud-Computing-System verwendet. Anwender können darauf ihre eigenen Services entwickeln und dem System Ziele vorgeben. Weitere Anwendungs-Szenarien für OCμ Techniken finden sich in wirtschaftlichen Organisationen oder der Raumfahrt. Innerhalb einer Firma können beispielsweise einzelnen Rechner zusammengeschlossen werden, um die verfügbare Rechenkapazität zu erhöhen. Der Verwaltungsaufwand kann dank der Selbst-X Eigenschaften gering gehalten werden. Bei unbemannten Expeditionen zu entfernten Planeten können die Selbst-X Eigenschaften eine große Erleichterung sein, da kein Administrator vorhanden ist, um mögliche Probleme zeitnah zu lösen.

In dieser Arbeit wurde der Organic Manager vorgestellt, der die Selbst-Konfiguration, Selbst-Optimierung und Selbst-Heilung umsetzt. Kapitel 4 beschreibt einen zweistufigen Ansatz, der die Planungsphase des Organic Managers implementiert. Der Planner Manager setzt die Selbst-X Eigenschaften mit Hilfe eines automatischen Planers um. Für diesen Planer wurden mehrere Modelle entwickelt, die verschiedene Mächtigkeiten und Fähigkeiten besitzen.

Um die Reaktionszeit des Systems zu verkürzen, wurde ein Reflex Manager entwickelt. Er speichert Pläne des Planers und kann sie wiederverwenden. Der dabei beobachtete Systemzustand wird als Schlüssel verwendet. Dabei ist hervorzuheben, dass der Reflex Manager nicht nur Pläne erneut verwenden kann, wenn exakt der gleiche Zustand auftritt. Er kann auch Pläne nutzen, die ursprünglich für einen anderen, ähnlichen Zustand erstellt wurden. Hierfür wurden zwei Metriken entwickelt, die Systemzustände vergleichen können.

Der Actuator ist für die Ausführung von Plänen zuständig. Durch die zweistufige Planungsphase erhält er Pläne vom Planner Manager wie auch vom Reflex Manager. Er kann die Ausführung eines Planes unterbrechen und verschiedene Pläne

vergleichen. Je nach Ergebnis des Vergleichs kann er anschließend den auszuführenden Plan wechseln.

Vorgestellt wurden ein Boolesches und Numerisches Modell. Das Boolesche Modell hat eine geringere Planungskomplexität als das Numerische. Wie sich jedoch gezeigt hat, ist es einem Numerischen Modell unterlegen: Das Boolesche Modell benötigt mehrere Planungsschritte um den gewünschten Zustand zu erreichen, wohingegen das Numerische Modell meist nur einen Planungsschritt braucht. Zwischen den Planungsschritten muss ausreichend Zeit sein, um die Veränderungen des Systems zu beobachten und die Informationen in den Knoten zu sammeln. Deshalb benötigt dass Boolesche Modell insgesamt länger zum Erreichen des Zielzustandes als das Numerische Modell.

In Kapitel 5 wurde ein Cloud-Computing-Szenario vorgestellt, in dem OCμ als Plattform dient. Die Selbst-Heilung, Selbst-Optimierung und Selbst-Konfiguration werden dabei gemeinsam umgesetzt und evaluiert. Wie sich zeigt, kann ein automatischer Planer mit dem Numerischen Modell alle drei Selbst-X Eigenschaften sowie die Ziele der Anwender zuverlässig und zeitnah umsetzen. Selbst bei häufigen Service-Ausfällen kann ein gutes Ergebnis erreicht werden.

Das Numerische Modell wurde in Kapitel 6 erweitert und ebenfalls anhand des Cloud-Computing-Szenarios evaluiert. Wie auch das Numerische Modell erfüllt es die geforderten Selbst-X Eigenschaften und kann häufige Service-Ausfälle abfangen. Zudem bietet es den Anwendern die Möglichkeit, Abhängigkeiten zwischen Services zu definieren. Wie gezeigt wurde, können diese Abhängigkeiten auch auf verschiedene Knoten verteilt werden. In beiden Fällen kann der Organic Manager die Abhängigkeiten selbstständig erfüllen.

In weiteren Szenarien wurden in Kapitel 7 die Wartung und der Ausfall von Knoten simuliert. Durch vorausschauendes Handeln kann der Organic Manager das System vorbereiten, so dass eine Wartung keine Beeinträchtigung der Anwender verursacht. Die vom automatischen Planer umgesetzte Selbst-Heilung sorgt nach Ausfall eines Knotens für einen schnellen Neustart der betroffenen Services.

In diesen Evaluierungen war der Reflex Manager immer schneller als der Planner Manager und konnte so die Reaktionszeit des Systems verkürzen. Anhand einer Simulation wurden die verschiedenen Parameter des Reflex Managers untersucht, um ihren Einfluss auf seine Trefferrate zu analysieren. Die Verdrängungsstrategie kann die Trefferrate wesentlich erhöhen. Sie sollte je nach System-Verhalten gewählt werden.

8.2 Ausblick

Basierend auf OCμ wurde die Trust Enabeling Middleware (TEM) [18] entwickelt. Sie bietet eine Möglichkeit das bestehende System durch Vertrauenswerte zu erweitern. Gefördert wird dieses Projekt im Rahmen des DFG Projekts OC-Trust. Kiefhaber et al. [16], [17] berechnen beispielsweise die Zuverlässigkeit von Knoten und können so zur Laufzeit partiell defekte Knoten erkennen.

Wenn ein neuer Knoten ein Netz betritt, besitzt er noch kein Wissen über die Zuverlässigkeit der anderen Knoten. In einem solchen Fall kann er benachbarte Knoten um ihre Meinung bitten und so die Reputation [15] eines Knoten in Erfahrung bringen. Weitere Verfahren ermöglichen eine Einschätzung der Zuverlässigkeit der berechneten Vertrauenswerte [14].

Diese verschiedenen Vertrauenswerte können verwendet werden, um die Umsetzung der Selbst-X Eigenschaften zu verbessern. In dieser Arbeit wurden beispielsweise Knoten mit gleicher Auslastung und Ressourcen als gleich gut geeignet für den Start eines Services angesehen. Wenn dieser Service jedoch besonders wichtig ist, möchte man ihn nur auf vertrauenswürdigen Knoten laufen lassen.

In der vorliegenden Arbeit wird manuell ausgewählt, welche Verdrängungsstrategie der Reflex Manager verwenden soll. Hier wäre es interessant zu erforschen, ob eine geeignete Verdrängungsstrategie automatisch ausgewählt und während der Laufzeit ausgetauscht werden kann (vgl. [31]). Während der Laufzeit könnte man ausgewählte Zustände speichern und im Zeitverlauf analysieren, ob sich ein zeitliches Muster entdecken lässt. Wenn beispielsweise ein zyklisches Verhalten erkannt wird, könnte man auf die Verdrängungsstrategie Similar wechseln.

In dieser Arbeit wurden zwei Metriken vorgestellt um Zustände zu vergleichen. In den Evaluierungen wurde jedoch die euklidische Metrik verwendet. Für die gewichtete Metrik müssen die Gewichte explizit zur Designzeit vorgegeben werden. Anwender können also zur Laufzeit keine neuen Service-Typen einbringen. Hier wäre es interessant, eine Regelschleife zwischen der Metrik und dem Reflex Manager einzuführen und die Gewichte im Lauf der Zeit lernen zu lassen.

Dazu muss getestet werden, ob der Plan des Planers bereits im Reflex Manager gespeichert ist und welcher Zustand verwendet wurde. Anschließend könnte man die Zustände vergleichen und herausfinden, welche Werte einen großen Einfluss auf die Metrik haben. Die Gewichtung dieser Werte könnte gesenkt werden. Wenn der Reflex Manager einen falschen Plan vorschlägt, könnten einzelne Gewichte erhöht werden.

Durch die Flexibilität des Planers kann OCμ auch in anderen Szenarien eingesetzt werden. Denkbar wäre es den Anwendern bzw. Services über eine Schnittstelle die Fähigkeiten des Planers anzubieten. Dann könnten sie ihre Planungsprobleme von dem automatischen Planer lösen lassen.

Lebenslauf von Julia Schmitt

Persönliche Daten

Name:	Julia Helene Schmitt
Geburtsdatum:	30.12.1983
Geburtsort:	Augsburg

Ausbildung

1996 – 2000	Realschule Bobingen Abschluss: Mittlere Reife
2000 – 2002	Lechwerke AG, Augsburg Ausbildung zur Industriekauffrau
2002 – 2004	Berufsoberschule Augsburg Abschluss: Allgemeine Hochschulreife
2004 – 2009	Studium Wirtschaftsmathematik an der Universität Augsburg Abschluss: Diplom-Wirtschaftsmathematik Thema der Diplomarbeit: „Automatische Erkennung von Systemausfällen mit Hilfe von Hidden Markov Modellen und Support Vector Machines“
seit 2009	Wissenschaftliche Angestellte am Lehrstuhl für Systemnahe Informatik und Kommunikationssysteme, Universität Augsburg

Veröffentlichungen als Erstautorin

- Using an Automated Planner to Control an Organic Middleware
  Julia Schmitt, Michael Roth, Rolf Kiefhaber, Florian Kluge, Theo Ungerer
  Proceedings of the Fifth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2011), Seite 71-78

- Concept of a Reflex Manager to Enhance the Planner Component of an Autonomic/Organic System
  Julia Schmitt, Michael Roth, Rolf Kiefhaber, Florian Kluge, Theo Ungerer

Proceedings of the 8th International Conference on Autonomic and Trusted Computing (ATC 2011), Seite 19-30

- Realizing Self-x Properties by an Automated Planner
  Julia Schmitt, Michael Roth, Rolf Kiefhaber, Florian Kluge, Theo Ungerer
  Poster of the 8th International Conference on Autonomic Computing (ICAC 2011)

Weitere Veröffentlichungen

2011

- The Neighbor-Trust Metric to Measure Reputation in Organic Computing Systems
  Rolf Kiefhaber, Stephan Hammer, Benjamin Savs, Julia Schmitt, Michael Roth, Florian Kluge, Elisabeth André, Theo Ungerer
  Proceedings of the 2nd Workshop on Trustworthy Self-Organizing Systems (TSOS 2011)

- Organic Computing Middleware for Ubiquitous Environments
  Michael Roth, Julia Schmitt, Rolf Kiefhaber, Florian Kluge, Theo Ungerer
  Organic Computing - A Paradigm Shift for Complex Systems, Seite 339-351

2010

- Trust Measurement Methods in Organic Computing Systems by Direct Observation
  Rolf Kiefhaber, Benjamin Satzger, Julia Schmitt, Michael Roth, Theo Ungerer
  8th International Conference on Embedded and Ubiquitous Computing (EUC 2010)

- The Delayed Ack Method to Measure Trust in Organic Computing Systems
  Rolf Kiefhaber, Benjamin Satzger, Julia Schmitt, Michael Roth, Theo Ungerer
  Trustworthy Self-Organizing Systems (TSOS) - Workshop at the Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO) 2010

9 Literaturverzeichnis

[1] Organic Computing Initiative. `http://www.organic-computing.de/spp`, 2011.

[2] JXTA Project. `http://jxta.kenai.com/`, 2012.

[3] JavaFF. http://personal.cis.strath.ac.uk/ ac/JavaFF/, Nov. 2011.

[4] R. Allrutz, C. Cap, S. Eilers, D. Fey, H. Haase, C. Hochberger, W. Karl, B. Kolpatzik, J. Krebs, F. Langhammer, P. Lukowicz, E. Maehle, J. Maas, C. Müller-Schloer, R. Riedl, B. Schallenberger, V. Schanz, H. Schmeck, D. Schmid, W. Schröder-Preikschat, T. Ungerer, H.-O. Veiser und L. Wolf. VDE/ITG/-GI - Positionspapier Organic Computing: Computer- und Systemarchitektur. *Gesellschaft für Informatik e.V.*, 2003.

[5] F. Bagci, F. Kluge, B. Satzger, A. Pietzowski, W. Trumler und T. Ungerer. Experiences with a Smart Office Project. *Research in Mobile Intelligence*, Seite 294, 2010.

[6] R. Fikes und N. J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.

[7] M. Ghallab, E. Nationale, C. Aeronautiques, C. K. Isi, S. Penberthy, D. E. Smith, Y. Sun und D. Weld. PDDL - The Planning Domain Definition Language. Technical report, 1998.

[8] E. Goldberg und Y. Novikov. BerkMin: A fast and robust Sat-solver. *Discrete Applied Mathematics*, 155(12):1549 – 1561, 2007.

[9] J. Hoffmann und B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[10] J. H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1975.

[11] P. Horn. Autonomic Computing: IBMs Perspective on the State of Information Technology also known as „IBM's Autonomic Computing Manifesto". *IBM Corporation*, Seite 1–39, 2001.

[12] A. Iwasaki, T. Nozoe, T. Kawauchi und M. Okamoto. Design of Bio-inspired Fault-tolerant Adaptive Routing Based on Enzymatic Feedback Control in the Cell: Towards Averaging Load Balance in the Network. In *Frontiers in the Convergence of Bioscience and Information Technologies*, FBIT, Seite 845 –850, 2007.

[13] J. O. Kephart und D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, 2003.

[14] R. Kiefhaber, G. Anders, F. Siefert, T. Ungerer und W. Reif. Confidence as a Means to Assess the Accuracy of Trust Values. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, Seite 690 –697, Juni 2012.

[15] R. Kiefhaber, S. Hammer, B. Savs, J. Schmitt, M. Roth, F. Kluge, E. Andre und T. Ungerer. The Neighbor-Trust Metric to Measure Reputation in Organic Computing Systems. In *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2011 Fifth IEEE Conference on*, Seite 41 –46, Okt. 2011.

[16] R. Kiefhaber, B. Satzger, J. Schmitt, M. Roth und T. Ungerer. The Delayed Ack Method to Measure Trust in Organic Computing Systems. In *Self-Adaptive and Self-Organizing Systems Workshop (SASOW), 2010 Fourth IEEE International Conference on*, Seite 184 –189, Sept. 2010.

[17] R. Kiefhaber, B. Satzger, J. Schmitt, M. Roth und T. Ungerer. Trust Measurement Methods in Organic Computing Systems by Direct Observation. In *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on*, Seite 105 –111, Dez. 2010.

[18] R. Kiefhaber, F. Siefert, G. Anders, T. Ungerer und W. Reif. The Trust-Enabling Middleware: Introduction and Application. Technical Report 2011-10, Informatik, 2011.

[19] F. Kluge. *Autonomic- und Organic-Computing-Techniken für eingebettete Echtzeitsysteme*. Doktorarbeit, Universität Augsburg, 2011.

[20] F. Kluge, S. Uhrig, J. Mische und T. Ungerer. A Two-Layered Management Architecture for Building Adaptive Real-Time Systems. In *Software Technologies for Embedded and Ubiquitous Systems*, Band 5287 der *Lecture Notes in Computer Science*, Seite 126–137. Springer Berlin / Heidelberg, 2008.

[21] M. Mamei und F. Zambonelli. Spatial Computing: The TOTA Approach. In *Self-star Properties in Complex Information Systems, Bertinoro Workshops*, Band 3460 der *Lecture Notes in Computer Science*, Seite 307–324. Springer Berlin / Heidelberg, 2005.

[22] F. Mösch, M. Litza, A. Auf, E. Maehle, K. Großpietsch und W. Brockmann. ORCA - Towards an Organic Robotic Control Architecture. In *Self-Organizing Systems*, Band 4124 der *Lecture Notes in Computer Science*, Seite 251–253. Springer Berlin / Heidelberg, 2006.

[23] C. Müller-Schloer. Organic computing: on the feasibility of controlled emergence. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '04, Seite 2–5. ACM, 2004.

[24] F. Nafz, H. Seebach, J.-P. Steghöfer, G. Anders und W. Reif. Constraining Self-organisation Through Corridors of Correct Behaviour: The Restore Invariant Approach. In *Organic Computing - A Paradigm Shift for Complex Systems*, Band 1 der *Autonomic Systems*, Seite 79–93. Springer Basel, 2011.

[25] B. Nebel und J. Koehler. Plan Reuse versus Plan Generation: A Theoretical and Empirical Analysis. *Artificial Intelligence*, 76(1-2):427–454, Juli 1995.

[26] M. Nickschas und U. Brinkschulte. CARISMA - A Service-Oriented, Real-Time Organic Middleware Architecture. *Journal of Software*, 4(7):654–663, 2009.

[27] E. P. D. Pednault. ADL: exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, Seite 324–332, 1989.

[28] A. Pietzowski. *Selbstschutz in Organic- und Ubiquitous-Middleware-Systemen unter Verwendung von Computer-Immunologie*. Doktorarbeit, Universität Augsburg, 2008.

[29] A. Pietzowski, B. Satzger, W. Trumler und T. Ungerer. Using Positive and Negative Selection from Immunology for Detection of Anomalies in a Self-Protecting Middleware. In *INFORMATIK 2006 – Informatik für Menschen*, Band P-93 der *GI-Edition – Lecture Notes in Informatics*, Seite 161–168, Bonn, Germany, Sept. 2006. Köllen Verlag.

[30] H. Prothmann, F. Rochner, S. Tomforde, J. Branke, C. Müller-Schloer und H. Schmeck. Organic Control of Traffic Lights. In *Autonomic and Trusted Computing*, Band 5060 der *Lecture Notes in Computer Science*, Seite 219–233. Springer Berlin / Heidelberg, 2008.

[31] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely und J. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, Seite 381–391. ACM, 2007.

[32] U. Richter, M. Mnif, J. Branke, C. Müller-Schloer und H. Schmeck. Towards a generic observer/controller architecture for Organic Computing. In *GI Jahrestagung (1)*, Seite 112–119, 2006.

[33] F. Rochner, H. Prothmann, J. Branke, C. Müller-Schloer und H. Schmeck. An Organic Architecture for Traffic Light Controllers. In *Informatik 2006 – Informatik für Menschen*, Band P-93 der *Lecture Notes in Informatics (LNI)*, Seite 120–127. Köllen Verlag, Okt. 2006.

[34] M. Roth, J. Schmitt, R. Kiefhaber, F. Kluge und T. Ungerer. Organic Computing Middleware for Ubiquitous Environments. In *Organic Computing - A Paradigm Shift for Complex Systems*, Seite 339–351. Springer Basel, 2011.

[35] B. Satzger. *Self-healing distributed systems*. Doktorarbeit, Universität Augsburg, Germany, 2008.

[36] B. Satzger, A. Pietzowski, W. Trumler und T. Ungerer. A Lazy Monitoring Approach for Heartbeat-Style Failure Detectors. *Availability, Reliability and Security, International Conference on*, 0:404–409, 2008.

[37] B. Satzger, A. Pietzowski, W. Trumler und T. Ungerer. Using automated planning for trusted self-organising organic computing systems. *Autonomic and Trusted Computing*, Seite 60–72, 2010.

[38] H. Schmeck. Organic Computing - a New Vision for Distributed Embedded Systems. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, Seite 201 – 203, Mai 2005.

[39] J. Schmitt, M. Roth, R. Kiefhaber, F. Kluge und T. Ungerer. Concept of a Reflex Manager to Enhance the Planner Component of an Autonomic / Organic System. In *Autonomic and Trusted Computing*, Band 6906 der *Lecture Notes in Computer Science*, Seite 19–30. 2011.

[40] J. Schmitt, M. Roth, R. Kiefhaber, F. Kluge und T. Ungerer. Realizing Self-x Properties by an Automated Planner. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC 2011, Seite 185–186, 2011.

[41] J. Schmitt, M. Roth, R. Kiefhaber, F. Kluge und T. Ungerer. Using an Automated Planner to Control an Organic Middleware. In *Fifth International Conference on Self-Adaptive and Self-Organizing Systems*, Seite 71–78, 2011.

[42] H. Seebach. *Konstruktion selbst-organisierender Softwaresysteme*. Doktorarbeit, Universität Augsburg, 2011.

[43] W. Trumler. *Organic Ubiquitous Middleware*. Doktorarbeit, Universität Augsburg, 2006.

[44] W. Trumler, J. Ehrig, A. Pietzowski, B. Satzger und T. Ungerer. A Distributed Self-healing Data Store. In *Autonomic and Trusted Computing*, Band 4610 der *Lecture Notes in Computer Science*, Seite 458–467. Springer Berlin Heidelberg, 2007.

[45] W. Trumler, R. Klaus und T. Ungerer. Self-configuration via Cooperative Social Behavior. In *Third International Conference, ATC*. Springer-Verlag Berlin Heidelberg, 2006.

[46] W. Trumler, J. Petzold, F. Bagci und T. Ungerer. AMUN: an autonomic middleware for the Smart Doorplate Project. *Personal and Ubiquitous Computing*, 10(1):7–11, 2006.

[47] W. Trumler, A. Pietzowski, B. Satzger und T. Ungerer. Adaptive Self-optimization in Distributed Dynamic Environments. *Self-Adaptive and Self-Organizing Systems, 2007. SASO '07. First International Conference on*, Seite 320 –323, Juli 2007.

[48] A. von Renteln und U. Brinkschulte. Implementing and Evaluating the AHS Organic Middleware - A First Approach. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*, Seite 163 –169, Mai 2010.

10 Abbildungsverzeichnis